1. Preface
    1. Jy0030: Java OOP: Preface to ITSE 2317
2. Essence of OOP
    1. Java1632 The Essence of OOP using Java, Static Initializer Blocks
    2. Java1634 The Essence of OOP using Java, Instance Initializers
    3. Java1636 Java OOP Member Classes
    4. Java1638 Java OOP Local Classes
    5. Java1640 Java OOP Anonymous Classes
    6. Java1642 The Essence of OOP using Java, Nested Top-Level Classes
3. Generics
    1. Java4210: Getting Started with Generics
    2. Java4210r: Review of Getting Started
    3. Java4220r: Review of Generic Types
    4. Java4230r: Review of Generic Methods
    5. Java4240r: Review of Bounded Type Parameters
    6. Java4250r: Review of Generics, Inheritance, and Subtypes
    7. Java4260r: Review of Type Inference
    8. Java4270r: Review of Wildcards
4. Event Handling
    1. Preface to Event Handling
        1. Jy0035: Java OOP: Preface to Event Handling
    2. Java Event Handling Basics
        1. Java0073 Java OOP The AWT and Swing, A Preview
        2. Java0077 Java OOP Callbacks - I
        3. Java0078 Java OOP Callbacks - II
        4. Java0079 Java OOP Callbacks - III
        5. Java0080 Java OOP Event Handling in JDK 1.1, A First Look, Delegation Event Model

6. Java0081 Java OOP Swing and the Delegation Event Model
7. Java082 Sharing a Listener Object Among Visual Components
8. Java084 Low-level and Semantic Events
9. Java085 Swing, New Event Types in Swing
10. Java087 Swing, Understanding getContentPane() and other JFrame Layers
11. Java090 Scrollbar (Adjustment) Event Handling
12. Java092 Mouse Motion Events
13. Java094 Using Abbreviated Inner Classes
14. Java095 Container Events and More on Inner Classes
15. Java096 Program-Generated Events and the AWTEventMulticaster Class
16. Java097 Component Events
17. Java099 Item Events
18. Java100 Creating, Trapping, and Processing Custom Event Types
19. Java101 Paint Events
20. Java102 Handling Events in Extended Components without Listener Objects
21. Java104 Posting Synthetic Events to the System Event Queue
22. Java105 Text Events
23. Java107 Understanding Action Objects in Java
24. Java110 The AWT Package, An Overview
25. Java112 Placing Components in Containers, Absolute Coordinates
26. Java114 BorderLayout
27. Java116 FlowLayout
28. Java118 GridLayout
29. Java120 CardLayout

30. Java1855 The KeyEventDispatcher in Java
31. Java1856 The KeyEventPostProcessor in Java (Capturing Keyboard Strokes in Java)

3. Event Handling with Graphics
    1. Part 1
        1. Java3102 Java OOP Modifying the World and SimpleTurtle Classes
        2. Java3104 Java OOP Modifications to the Turtle and SimpleTurtle Classes
        3. Java3106 Java OOP Incorporating GUI Components into a World Object
        4. Java3108 Java OOP Background Color, Text Color, Mouse Clicks, etc
        5. Java3110 Java OOP Panels, Labels, Text Fields, and Buttons
    2. Part 2
        1. Java3112 Java OOP Using Alpha Transparency with Ericson's Media Library
        2. Java3114 Java OOP Controlling Opacity with a Slider
        3. Java3116 Java OOP Controlling an Edge Detector with a Slider
        4. Java3118 Java OOP Controlling an Image-Scaling Program with a Slider
        5. Java3120 Java OOP Controlling Image Rotation with a Slider and Affine Transforms
    3. Part 3
        1. Java3122 Java OOP Opening an Image File in a PictureExplorer Object
        2. Java3124 Java OOP Extracting pixel color data from a PictureExplorer object

        3. Java3126 Java OOP Handling document events on a text field and creating a color swatch
        4. Java3128 Java OOP Using a JColorChooser object
        5. Java3130 Java OOP A Pixel Color Editor
    4. Practice Programs
        1. Java OOP: ITSE 2317 Practice Programs 1
        2. Java OOP: ITSE 2317 Practice Programs 2
        3. Java OOP: ITSE 2317 Practice Programs 3
4. Review Event Handling
    1. Jy0037: Review Event Handling
5. JavaServer Pages (JSP)
    1. Java4309-Links to JSP Learning Resources

Jy0030: Java OOP: Preface to ITSE 2317
This module serves as the Preface to material for the course, ITSE2317 - Java Programming (Intermediate), which Prof. Baldwin teaches at Austin Community College in Austin, TX.

Revised: Fri Sep 23 13:08:46 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Welcome
- Essence of OOP
- Downloads
- Miscellaneous

Welcome

Welcome to the course material for ***ITSE2317 - Java Programming (Intermediate)*** , which I teach at Austin Community College in Austin, TX.

Information about the course

The college website for this course is: http://www.austincc.edu/baldwin/

As of December 2013, the State of Texas description for this course reads as follows:

Note: *"Introduction to object-oriented Java programming including the fundamental syntax and semantics of Java for applications and web applets."*

The prerequisite for the course is ITSE 2321 - Object-Oriented Programming (Java) or department approval.

In the event that you enroll in this course without having successfully completed a section of that prerequisite course taught by Prof. Baldwin, or you successfully completed that course under Prof. Baldwin but a significant amount of time has elapsed since you completed the course, you should review the material in the following three Ebooks to make certain that you understand all of the material covered in those Ebooks before embarking on this course:

- Programming Fundamentals with Java
- Java OOP Self-Assessment
- ITSE 2321 - Object-Oriented Programming (Java)

The course covers **three major topics** :

- Generics
- Event Handling
- JavaServer Pages (JSP)

Essence of OOP

In addition to the three major topics listed above , there is also a topic named **Essence of OOP** that contains necessary background material for an understanding of **Event Handling** . This is a continuation of a topic having the same name that begins in the prerequisite course, ITSE 2321 - Object-Oriented Programming (Java) .

Downloads

I encourage you to take advantage of the download options that OpenStax has to offer in order to customize this material for use in your organized courses or for personal self study.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jy0030: Java OOP: Preface to ITSE 2317
- File: Jy0030.htm
- Published: 11/29/12

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the printed PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1632 The Essence of OOP using Java, Static Initializer Blocks
Baldwin explains and illustrates the use of static initializer blocks.

Revised: Wed May 11 14:42:52 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled ITSE2317 - Java Programming (Intermediate) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1632 The Essence of OOP using Java, Static Initializer Blocks
- File: Java1632.cnx.htm
- Published: 01/13/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1634 The Essence of OOP using Java, Instance Initializers
Baldwin explains the use of instance initializers, static initializers in conjunction with constructors, and the initialization of ordinary instance variables. He explains and demonstrates the order in which they are executed.

Revised: Wed May 11 14:51:26 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled ITSE2317 - Java Programming (Intermediate) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If

not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1634 The Essence of OOP using Java, Instance Initializers
- File: Java1634.cnx.htm
- Published: 01/13/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on

Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1636 Java OOP Member Classes
Baldwin explains the various relationships that exist among member classes and their enclosing classes.

Revised: Wed May 11 15:16:30 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
    - What can you include in a class definition?
    - What is a member class?
    - What about a member interface?
    - Why use member classes?
    - Smoke and mirrors
- Discussion and sample code
    - Class containment hierarchy
    - Behavior of the program
    - Structure of the program
- Run the program
- Summary
- What's next?
- Complete program listing

- Miscellaneous
- Figures
- Listings

Preface

This module is one in a collection of modules designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

(Editor's note: As you read this module, you will see that it was originally written around 2003. However, despite many improvements in Java since then, most of what was true then is still true in 2013.)

This module makes several references to my website, which is located at http://www.dickbaldwin.com/toc.htm .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1 . Class files produced when the program is compiled.
- Figure 2 . Screen output.
- Figure 3 . Screen output.
- Figure 4 . Screen output.
- Figure 5 . Screen output.
- Figure 6 . Screen output.
- Figure 7 . Screen output.
- Figure 8 . Screen output.
- Figure 9 . Screen output.

- Figure 10 . Screen output.
- Figure 11 . Screen output.
- Figure 12 . Screen output.
- Figure 13 . Screen output.
- Figure 14 . Screen output.
- Figure 15 . Screen output.
- Figure 16 . Screen output.

Listings

- Listing 1 . The main method.
- Listing 2 . The class named X.
- Listing 3 . Beginning of the top-level class named A.
- Listing 4 . Constructor for class A.
- Listing 5 . Beginning of the member class named B.
- Listing 6 . Constructor for class B.
- Listing 7 . Beginning of the member class named C.
- Listing 8 . Beginning of constructor for class C.
- Listing 9 . The private member class named D.
- Listing 10 . More constructor code for class C.
- Listing 11 . More constructor code for class C.
- Listing 12 . The method named bShow.
- Listing 13 . The method named aShow.
- Listing 14 . Beginning of the cShow method.
- Listing 15 . More cShow method code.
- Listing 16 . Call the aShow method.
- Listing 17 . More cShow method code.
- Listing 18 . More cShow method code.
- Listing 19 . More cShow method code.
- Listing 20 . More cShow method code.
- Listing 21 . More cShow method code.
- Listing 22 . More cShow method code.
- Listing 23 . Overridden toString method in class C.
- Listing 24 . More cShow method code.
- Listing 25 . Complete program listing.

Preview

What can you include in a class definition ?

There are several different kinds of items that can be contained in a class definition. As you learned in the early modules in this series, the list includes:

- Static variables
- Instance variables
- Static methods
- Instance methods
- Constructors

As you can learn at http://www.dickbaldwin.com/toc.htm , the list also includes:

- Static initializer blocks
- Instance initializers

Can also contain other class definitions

In this and the next two modules, you will learn that a class definition can also contain the following three kinds of ***inner classes*** *:*

- Member classes
- Local classes
- Anonymous classes

A class can also contain **Nested top-level classes and interfaces** , which you can learn about at http://www.dickbaldwin.com/toc.htm .

> *(Note that it is questionable whether a nested top-level class or interface should be referred to as an inner class, because an object of a nested top-level class can exist in the absence of an object of the enclosing class. Regardless of whether the term*

> *inner class applies, a nested top-level class is defined within the definition of another class, so its definition is internal to the definition of another class.)*

This module will be dedicated to an explanation of **member classes** . Subsequent modules will explain the other two types of inner classes in the above list .

What is a member class ?

A *member class* is a class that is defined inside the definition of another class, *(without the use of the static modifier as is the case with a nested top-level class).*

An object of the member class must be internally linked to an object of the enclosing class, *(which is not the case with a nested top-level class).*

Thus, a member class is truly an inner class. *(An object of the member class cannot exist in the absence of an object of the enclosing class.)*

What about a member interface ?

Interfaces defined within classes are implicitly static. This means that they are always *top-level*. There is no such thing as a member interface, a local interface, or an anonymous interface.

Why use member classes ?

Probably the most important benefit of member classes has to do with accessing the other members of enclosing classes. The methods of a member class have direct access to all the members of the enclosing classes,

including private members. Thus the use of member classes can eliminate the requirement to connect objects together via constructor parameters.

This is particularly useful in those cases where there is no reason for an object of a member class to exist in the absence of an object of the enclosing class, and where the methods of the object of the member class need access to members of the object of the enclosing class.

Data structures and iterators

For example, there is usually no reason for an **Iterator** object to exist in the absence of the data-structure object for which it is designed to provide iterator services. Also, the iterator object usually needs to have ready access to the members of the data-structure object, some or all of which may be private. Thus, a class from which an **Iterator** object can be constructed is a good candidate for inclusion as a member class in the class from which the associated data-structure object is instantiated.

Listener objects

Another common use for inner classes is in the definition of classes from which listener objects *(which listen for events fired by other objects)* are instantiated. *(However, it may be more common to use anonymous classes than member classes for this purpose.)*

What does Flanagan have to say?

Here is how David Flanagan, author of Java in a Nutshell, summarizes his discussion of member classes.

> *"A class defined as a member (non-static) of another. Each instance has an enclosing instance, and can use its members. New syntax for **this** , **new** , and **super** . Cannot have **static** members. Cannot have same name as containing class."*

According to Flanagan, the main features of member classes are:

- Every instance of a member class is internally associated with an instance of the class that defines or contains the member class.
- The methods of a member class can implicitly refer to the fields defined within the member class, as well as those defined by any enclosing class, including **private** fields of the enclosing class.

Smoke and mirrors

Every class definition in a Java program, including nested top-level classes, member classes, local classes, and anonymous classes, produces a class file when the program is compiled. According to Flanagan,

> *"The Java Virtual Machine knows nothing about nested top-level classes and interfaces or the various types of inner classes. Therefore, the Java compiler must convert these new types into standard non-nested class files that the Java interpreter can understand. This is done through source code transformations that insert $ characters into nested class names. These source code transformations may also insert hidden fields, methods, and constructor arguments into the affected classes."*

A reference to the containing object

For example, the compiler automatically inserts a private instance variable in the member class to hold a reference to the containing object. It also inserts a hidden argument in all constructors for the member class, and passes the containing object's reference to the constructor for the member class. The modified constructor saves that reference in the private instance variable of the object of the member class. Thus each object instantiated from the member class contains a private reference to the containing object.

Accessing private members

In those cases where it is necessary for an object of the member class to access private members of the containing object, the compiler automatically creates and uses accessor methods that make such access possible.

Similar to your code

The bottom line is that the code that is automatically produced by the compiler is probably very similar to code that you would write if you were writing the program using only of top-level classes. The good news is that you don't have to write that extra code, and you don't have to maintain it. The extra code is written for you, and if you modify your class structure, the extra code is automatically modified accordingly.

Discussion and sample code

The paragraphs that follow will explain a program named **InnerClasses06** , which is designed specifically to illustrate various characteristics of member classes. I will discuss the program in fragments. A complete listing is shown in Listing 25 near the end of the module.

This program illustrates the use of member classes. The program consists of a total of six classes:

- Top-level classes named
    - InnerClasses06
    - A
    - X
- Member classes named
    - B
    - C
    - D

When compiled, the program produces the class files shown in Figure 1 .

Class containment hierarchy

Once you understand the class file naming convention, you can determine from the file names in Figure 1 that class B is a member class of class A.

> *(The class file named A$B.class indicates that the class named B is a member of the class named A.)*

Similarly, class C is a member of class B, and class D is a private member of class C.

> *(However, you cannot tell from the class file names that class D is private.)*

Behavior of the program

An object is instantiated from the class named A. This makes it possible to instantiate an object of the member class named B. The object of the class named B is internally linked to the object of the class named A.

> *(This causes the instance variable, constructor parameter, and accessor methods discussed above to be automatically created to link the object of the class named B to the object of the class named A.)*

The object of the class named B is used to instantiate an object of the member class named C. This object of the class C is linked to the object of the class named B.

Instantiate additional objects of classes A and B, plus an object of class D

When the object of the class named C is instantiated, the constructor for that class instantiates separate objects of the classes named A and B, and also instantiates an object of the **private** member class named D.

> *(We will see later that the new and separate object of the class named B continues to be internally linked to the original object of the Class named A, and is not internally linked to the new object of the class named A.)*

Instantiation of the object of class D illustrates the use of private member classes.

> *(Note that while top-level classes cannot be private, member classes can be private.)*

Perform a variety of operations

A variety of operations are performed from within the methods belonging to the object of the class C to illustrate the attributes and behavior of objects instantiated from member classes.

Comments in the code explain the purpose of each of those operations.

Many of those operations produce screen output, which will be shown in conjunction with the code that produced the output.

Structure of the program

The main method

The **main** method of the controlling class named **InnerClasses06** , is shown in Listing 1 .

The code in Listing 1 instantiates an object of the member class named C and calls the method named **cShow** on that object.

> *(Note that it is necessary to first instantiate objects of the enclosing classes named A and B before the object of the member class named C can be instantiated.)*

An independent top-level class named X

Listing 2 shows the definition of an independent top-level class named X.

This class will be extended by the class named C, which is a member of the class named B, which is a member of the class named A. This will illustrate that the inheritance hierarchy is independent of the containment hierarchy.

As you can see in Listing 2 , the class named X overrides the **toString** method to identify itself when called.

> *(The **toString** method is automatically called whenever an object's reference is passed as a parameter to the **println** method.)*

The top-level class named A

Listing 3 shows the beginning of the top-level class named A.

Listing 3 shows the declaration of two instance variables and three class variables in the class named A. All of the variables are private, and some

are initialized when declared. The three class variables will be used to maintain a count of the number of objects instantiated from the classes named A, B, and C.

> *(Because member classes **cannot contain static members** , the counter variables for the member classes named B and C were placed in the top-level class named A instead of placing them in their respective class definitions.)*

Constructor for class A

Listing 4 shows the constructor for the top-level class named A.

Whenever an object of the class named A is instantiated, the constructor does the following:

- Saves the value of an incoming parameter in a private instance variable named **aVar** .
- Increments the object counter named **objCntA** , maintaining a count of the objects instantiated from class A.
- Saves the value of the object counter in an instance variable named **objNumber** to identify the specific object.
- Displays a message showing the identification of the object being instantiated.

The screen output

The code in Listing 1 instantiates a new object of the class named A, passing the integer value 1 as a parameter to the constructor. As a result, the code in the constructor shown in Listing 4 produces the screen output shown in Figure 2 .

As you can see from the value of the object counter in Figure 2 , this is the first object instantiated from the class named A.

> *(The value passed, as a parameter to the constructor, is not displayed by the code in the constructor. That value will be displayed later.)*

The method named aShow

The class named A also defines a private method named **aShow** . I will defer my discussion of that method until later when it is called.

The member class named B

Listing 5 shows the beginning of the member class named B.

If you examine the complete listing of the program in Listing 25 , you will see that the class named B is defined internal to the class named A. In other words, the beginning of the definition of the class named B appears before the curly brace that signals the end of the definition of the class named A. Thus, the class named B is a member class of the class named A.

The code in Listing 5 declares two private instance variables and initializes one of them.

Constructor for class B

Listing 6 shows the entire constructor for the class named B.

Whenever an object of the class named B is instantiated, the constructor does the following:

- Saves the value of an incoming parameter in a private instance variable named **bVar** .
- Increments the object counter named **objCntB** , which is a class variable of the containing top-level class named A, maintaining a count of objects instantiated from class B.
- Saves the value of the object counter in an instance variable named **objNumber** to identify the specific object.

- Displays a message showing the identification of the object being instantiated.

The screen output

Listing 1 shows the instantiation of a new object of class B, immediately following the instantiation of an object of class A.

The object instantiated from the member class named B is linked to the object instantiated from the top-level class named A.

The constructors for the classes named A and B produce the two lines of output shown in Figure 3 , the first of which is a repeat of the output shown in Figure 2 .

The method named bShow

The class named B also defines a private method named **bShow** . As with the method named **aShow** mentioned earlier, I will defer a discussion of **bShow** until later when it is called.

The member class named C

Listing 7 shows the beginning of a member class named C.

Class C is a member of the class named B. In other words, the beginning of the definition of the class named C begins before the curly brace that ends the definition of the class named B.

The code in Listing 7 declares several instance variables for the class named C, and initializes two of them. The purpose of these variables will become clear later when they are used.

> *(Note also that class C extends class X, in order to illustrate that the class containment hierarchy is independent of the inheritance hierarchy.)*

Constructor for class C

Listing 8 shows the beginning of the constructor for the class named C.

Whenever an object of the class named C is instantiated, the constructor code shown in Listing 8 does the following:

- Saves the value of an incoming parameter in a private instance variable named **cVar** .
- Increments the object counter named **objCntC** , which is a class variable of the class named A, maintaining a count of objects instantiated from class C.
- Saves the value of the object counter in an instance variable named **objNumber** to identify the specific object.
- Displays a message showing the identification of the object being instantiated.

Screen output

Listing 1 shows the instantiation of a new object of class C, immediately following the instantiation of an object of class B.

The object instantiated from the member class named C is linked to the object instantiated from the member class named B. Similarly, the object instantiated from the member class named B is linked to the object instantiated from the top-level class named A.

The constructors for the classes named A, B, and C produce the three lines of output shown in Figure 4 , the first two of which are repeated from Figures 2 and Figure 3 .

The output shown in Figure 4 demonstrates that the code in Listing 1 causes the constructors for the three classes to be executed in sequence.

At this point, I am going to put the discussion of the class named C on hold and discuss another member class named D.

The private member class named D

Top-level classes cannot be private. However, member classes can be private provided that the using code is consistent with the use of private members. To demonstrate this, the class named C contains a private member class named D, which is shown in its entirety in Listing 9 .

The most significant thing about the class named D is that it is declared private.

When an object is instantiated from the class named D, it displays a couple of messages, one of which provides the name of the class file produced by the compiler to represent the class named D. We will see those messages shortly in conjunction with the instantiation of an object of the class named D.

Returning to the constructor for class C

Listing 10 shows the next statement in the constructor for the class named C. This statement instantiates an object of its private member class named D.

The code in Listing 10 causes the constructor for the class named D to be executed, producing the screen output shown in Figure 5 .

As mentioned earlier, comparing the class file name in Figure 5 with the class file naming convention for member classes, you can determine that D is a member of C, C is a member of B, and B is a member of A.

Instantiate independent objects of classes A and B

The remaining constructor code for class C is shown in Listing 11 .

The code in Listing 11 instantiates new and independent objects of the classes named A and B, both of which are enclosing classes of the member class named C.

> *(Note that the parameter values passed to the constructors are different than was the case for the objects instantiated in Listing 1 . We will see the result of that later.)*

I will display information about these two objects later. That information will show that the new object of the member class named B is linked to the original object of the enclosing class named A.

The screen output

In the meantime, when these two objects are instantiated, their constructors are executed, producing the screen output shown in Figure 6 .

In each case, the value of the object counter shows that this is the second object instantiated from each of these two classes.

Methods aShow, bShow, and cShow

The classes named A, B, and C, each contain display methods named **aShow** , **bShow** , and **cShow** respectively.

The method named **cShow** is rather long, and I will discuss it in detail shortly. For now, suffice it to say that code in **cShow** calls the private method named **bShow** in the containing object to which it is linked. Therefore, this will be an appropriate time to examine the method named **bShow** , which is defined in the member class named B.

The method named bShow

The **bShow** method, defined in the member class named B, is shown in Listing 12 . It is important to note that this is a private method.

When this method is called, it does the following:

- Displays the value of the constructor parameter passed to the object when it was constructed.
- Displays the identification of the object based on the value of the object counter when it was constructed.
- Calls the corresponding **aShow** method of the object of the containing class to which it is linked.

Since the code in the **bShow** method calls the private **aShow** method of the containing object to which it is linked, it is also time to take a look at that method.

The method named aShow

The **aShow** method, defined in the top-level class named A, is shown in Listing 13 . It is also important to note that this is a private method.

When this method is called, it does the following:

- Displays the value of the constructor parameter passed to the object when it was constructed.
- Displays the identification of the object based on the value of the object counter when it was constructed.

Containment hierarchy is displayed

Because **cShow** calls **bShow** , which in turn calls **aShow** , we should expect that the call to the **cShow** method on an object of the member class named C would display information about the containment hierarchy.

> *(Simply as another reminder, the containment hierarchy is completely independent of the inheritance hierarchy.)*

Calling cShow

Referring once more to Listing 1 , we see that the method named **cShow** is called on the object of the class named C when that object is instantiated. We will see the result of that call shortly.

The cShow method

Listing 14 shows the beginning of the **cShow** method.

The code in Listing 14

- Displays a string separator to help locate the specific output in the large quantity of output produced by the program.
- Displays the object identifier based on the object counter.
- Displays the value passed to the constructor when the object was instantiated.

The screen output

The code in Listing 14 produces the output shown in Figure 7 .

As you can see by comparing this with Listing 1 , this is the first object instantiated from the class named C, and is the object instantiated from the statement in the **main** method in Listing 1 . *(The constructor parameter value is 3.)*

Call the bShow method

Continuing with the code in the **cShow** method, the code in Listing 15 calls the private method named **bShow** on the containing object of the class B to which this object is linked.

As you will recall from the previous discussion, the code in the **bShow** method will, in turn, call the **aShow** method on the containing object of the class named A to which the object of the class B is linked.

The screen output

The code in Listing 15 produces the output shown in Figure 8 .

As you can see in Figure 8 , the linked objects of the classes B and A are the first objects instantiated from those classes. In addition, the saved values of the constructor parameters show that these are the objects that were instantiated by the statement in the **main** method of Listing 1 .

Call the aShow method

As I explained earlier, the object of the class C is linked to the containing object of the class named B. The code in Listing 16 shows that the object of the class C is also linked to the containing object of the class A *(even*

though the containing class named A is one level removed in the containment hierarchy).

Implicit access

The methods of a member class have implicit access to all members *(including private members)* of all containing classes. Thus, the code in the **cShow** method, belonging to the object of the class named C, can directly call the private **aShow** method of the containing class named A.

The screen output

Therefore, the code in Listing 16 produces the output shown in Figure 9 .

You can tell by the values displayed in Figure 9 that the **aShow** method called in Listing 16 was called on the same object on which the **aShow** method was called by the code in Listing 15 . However, in Listing 15 , the **bShow** method was called first, which in turn called the **aShow** method.

Accessing the object of the class C, and the this keyword

The syntax used with the keyword **this** is somewhat different for member classes and contained objects than is the case for top-level classes. For example, continuing with the method named **cShow** , the code in Listing 17 shows five different ways to access the object instantiated from the member class named C in order to get and display the name of the class file that represents the member class named C.

The screen output

All five statements in Listing 17 display the name of the same class file, as shown in Figure 10 .

Obviously in this situation, the last three statements in Listing 17 are overly complex. There is no particular problem writing code in the method named **cShow** to gain access to the object to which the method belongs. It isn't even necessary to use **this** to refer to that object, although the use of the hidden reference **this** may make the code more readable.

Accessing the containing object of the class B

However, things get a little more complicated when you need to gain access to a containing object, such as the containing object instantiated from the class named B.

The two statements shown in Listing 18 gain access to the containing object of the class named B. Each statement gets and displays the name of the class file that represents the member class named B. *(Note the use of the keyword **this** in these statements.)*

The screen output

The output produced by the code in Listing 18 is shown in Figure 11 . Once again, both statements get and display the name of the same class file.

Accessing the containing object of the class named A

Finally, the code in Listing 19 gains access to the containing object of the class named A. *(Once again, note the use of the **this** keyword in the statement in Listing 19 .)*

The code in Listing 19 produces the output shown in Figure 12 .

> *(Since the class named A is a top-level class, the name of the class file is the same as the name of the class, with no $ characters inserted by the compiler.)*

Investigate independent objects of classes A and B

Recall that when the object of the member class named C was instantiated, the constructor for the class instantiated independent objects of the enclosing classes named A and B, and saved those object's references in instance variables of the class named C.

> *(See Listing 11 , noting the parameter values of 10 and 20 passed to the constructors for A and B. Recall that the constructors for A and B save those parameter values in private instance variables named **aVar** and **bVar** .)*

Display variable values and class file names

The code in Listing 20 displays the values stored in the private instance variables belonging to those objects. The code in Listing 20 also displays the names of the class files representing the classes from which those objects were instantiated.

Screen output

The code in Listing 20 produces the output shown in Figure 13 .

There should be no surprises in the output shown in Figure 13 . The values of the instance variables match the parameter values passed to the constructors in Listing 11 when the objects were instantiated. The class file names match what you already know to be true from previous discussions earlier in this module.

Call the private bShow method

The code in Listing 21 is somewhat more interesting. This code calls the private **bShow** method on the separate object instantiated from the class named B in order to identify the object to which that object is linked.

The screen output

The output produced by the code in Listing 21 is shown in Figure 14 . Even though this object of the member class B was instantiated from within the constructor for the member class named C, the object of the class named B is internally linked to the object of the class named A that was originally used to instantiate the object of the class named C.

> *(See Listing 1 where the objects of classes named A, B, and C were originally instantiated. This object of the class named B is a different object from the object of the class named B instantiated in Listing 1 . This object of the class named B was instantiated by the code in Listing 11 .)*

How is this determined from Figure 14 ?

The second line in Figure 14 shows that a parameter value of 20 was received by the constructor when the object of the class named B was instantiated. This corresponds to the instantiation of the object by the code in the constructor in Listing 11 .

The third line in Figure 14 shows that this was the second object instantiated from the class named B. *(See the definition of the **bShow** method in Listing 12 , which displays the value stored in a variable that is used to save the object number.)*

The proof of the pudding

Now recall that the method named **bShow** *(belonging to an object of the class B)* calls the method named **aShow** belonging to the object of the class named A to which it is internally linked.

The fourth line in Figure 14 shows the value of the parameter passed to the constructor for the object of class A when that object was instantiated. *(See the definition of the **aShow** method in Listing 13 .)* This value corresponds to the value that was passed to the constructor for the original object of class A when it was constructed in Listing 1 .

> *(It does not correspond to the value passed to the constructor for the class named A when the object of the class A was constructed in Listing 11 .)*

The fifth line in Figure 14 shows that the object was the first object instantiated from the class named A.

Both B objects link to the same A object

Thus, both objects instantiated from the class named B in this program are internally linked to the same object instantiated from the class named A, which is the enclosing class for the class named B.

(However, had I instantiated the new object of the class B using a statement such as the following,

new A(100).new B(200).bShow();

the new object of the class B would have been linked to the new object of the class A rather than being linked to the original object of the class A. As you can see, keeping mental track of which object is linked to which other object could become complicated.)

Call the aShow method on the other A object

The code in Listing 22 calls the private **aShow** method on the independent object of the class A that was instantiated in the constructor for class C, shown in Listing 11 .

The output produced by the code in Listing 22 is shown in Figure 15 .

It should come as no surprise that this object of the class A was instantiated with a constructor parameter value of 10, and that it was the second object of the class named A that was constructed.

This is simply a matter of code in the constructor for class C instantiating an object of a top-level class, and is no different in concept from instantiating an object of the member class B, also shown in Listing 11 .

Inheritance and containment hierarchies are independent

The remaining code is designed to demonstrate that the containment hierarchy is completely independent of the inheritance hierarchy.

The class named C is a member of *(is contained in)* the class named B. That constitutes a part of the containment hierarchy.

The class named C also extends the class named X, which in turn extends the class named **Object** . That constitutes the inheritance hierarchy.

Overridden toString methods

I'm going to put the discussion of the method named **cShow** on hold and return to that discussion shortly.

The class named X inherits, and overrides the **toString** method, as shown in Listing 2 . When this version of the **toString** method is called, it returns the string " **toString in Class X** ".

The class named C, which extends the class named X, also overrides the **toString** method as shown in Listing 23 .

When this version of the **toString** method is called, it returns the string

" **toString in Class C** ".

We will see the impact of overriding these two methods later.

Illustrate the inheritance hierarchy

Returning to the **cShow** method, the code in Listing 24 illustrates the inheritance hierarchy to which the class named C belongs by getting and displaying the value stored in the instance variables named **className** belonging to the object instantiated from the class named C.

> *(The object contains two instance variables having the name **className** . One of these instance variables was contributed to the object by the superclass named X. The other was contributed to the object by the class named C.)*

Two instance variables named className

The **String** value **X** was stored in one of the instance variables named **className** by the initialization of the variable shown in Listing 2 .

The **String** value **C** was stored in the other instance variable named **className** by the initialization of the variable shown in Listing 7 .

> *(* Note that the variable named **className** is **protected** in the class named X. *A subclass method cannot access a private variable in a superclass. To be accessible by a subclass method, the superclass variable must be **protected** , **package-private** , or **public** .)*

Two overridden toString methods

An object instantiated from the class named C also contains two overridden versions of the **toString** method. One version of the method was contributed to the object by the superclass named X. The other version was contributed to the object by the class named C.

Call the toString methods

The code in Listing 24 also calls the two overridden **toString** methods belonging to the object instantiated from the classes named C. As explained earlier, one version of the **toString** method is overridden in the class named X and the other version is overridden in the class named C.

> *(Note the use of the **super** keyword to access the variable named **className** and the method named **toString** contributed to the object by the superclass named X.)*

The code in Listing 24 also signals the end of the **cShow** method.

The screen output

The output produced by the code in Listing 24 is shown in Figure 16 .

Figure 16 shows that even though the class named C is contained in the class named B, the superclass of C is X, and is not B. To repeat, the containment hierarchy is ***entirely independent*** of the inheritance hierarchy.

> *(Note, however, that there is nothing to prevent you from establishing an inheritance relationship between a member class and one of its containing classes if such a relationship will serve your needs. For example, in this program, it would be technically acceptable for the class named B to extend the class named A provided that either:*
>
> - *A noarg constructor is provided for the class named A, or*
> - *The constructor for the class named B calls the parameterized constructor belonging to the class named A.)*

And that is probably more than you ever wanted to know about the detailed relationships involving member classes. However, once you start using member classes, you will need to keep these relationships in mind.

Run the program

I encourage you to copy the code from Listing 25 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In addition to a number of other items, a class definition can contain:

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

This module explains member classes. Future modules will explain local classes and anonymous classes

A *member class* is a class that is defined inside the definition of another class without being declared static.

An object of the member class must be internally linked to an object of the enclosing class. A member class is truly an inner class because an object of the member class cannot exist in the absence of an object of the enclosing class.

The methods of a member class have direct access to all the members of the enclosing classes, including private members. Thus the use of member classes can eliminate the requirement to connect objects together via constructor parameters. This is particularly useful in those cases where there is no reason for an object of a member class to exist in the absence of an object of the enclosing class, and where the methods of the object of the member class need access to members of the object of the enclosing class.

The containment hierarchy of member classes is independent of the inheritance hierarchy. However, it is technically possible to establish an inheritance relationship between a member class and one of its enclosing classes.

Member classes may be declared private, and may be instantiated from code that would normally have access to a private member at that level.

What's next?

The next module in this series will explain and discuss local classes. A future module will explain *anonymous* classes.

Complete program listing

A complete listing of the program discussed in this module is show in Listing 25 .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Member Classes
- File: Java1636.htm
- Published: 11/18/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Figures

Figure 1 - Class files produced when the program is compiled.

```
A$B$C$D.class
A$B$C.class
A$B.class
A.class
InnerClasses06.class
X.class
```

Figure 2 - Screen output.

```
In xstr for A, objCntA = 1
```

Figure 3 - Screen output.

Figure 3 - Screen output.

```
In xstr for A, objCntA = 1
In xstr for B, objCntB = 1
```

Figure 4 - Screen output.

```
In xstr for A, objCntA = 1
In xstr for B, objCntB = 1
In xstr for C, objCntC = 1
```

Figure 5 - Screen output.

```
Construct obj of private class D.
Private class file name: A$B$C$D
```

Figure 6 - Screen output.

```
In xstr for A, objCntA = 2
In xstr for B, objCntB = 2
```

Figure 7 - Screen output.

```
-1-
In cShow, objNumber = 1
In cShow, cVar = 3
```

Figure 8 - Screen output.

```
-2-
In bShow, bVar = 2
In bShow, objNumber = 1
In aShow, aVar = 1
In aShow, objNumber = 1
```

Figure 9 - Screen output.

```
-3-
In aShow, aVar = 1
In aShow, objNumber = 1
```

Figure 10 - Screen output.

```
-4-
A$B$C
A$B$C
A$B$C
A$B$C
A$B$C
```

Figure 11 - Screen output.

```
-5-
A$B
A$B
```

Figure 12 - Screen output.

```
-6-
A
```

Figure 13 - Screen output.

```
-7-
In cShow, bVar = 20
A$B
In cShow, aVar = 10
A
```

Figure 14 - Screen output.

```
-8-
In bShow, bVar = 20
In bShow, objNumber = 2
In aShow, aVar = 1
In aShow, objNumber = 1
```

Figure 15 - Screen output.

```
-9-
In aShow, aVar = 10
In aShow, objNumber = 2
```

Figure 16 - Screen output.

```
-10-
className = C
toString in Class C
className = X
toString in Class X
```

Listings

Listing 1 - The main method.

Listing 1 - The main method.

```
public class InnerClasses06{
  public static void main(String[] args){
    new A(1).new B(2).new C(3).cShow();
  }//end main
}//end class InnerClasses06
```

Listing 2 - The class named X.

```
class X{//extends Object by default
  protected String className = "X";

  public String toString(){
    return "toString in Class X";
  }//end overridden toString
}//end class X
```

Listing 3 - Beginning of the top-level class named A.

Listing 3 - Beginning of the top-level class named A.

```
class A{
  private int aVar;
  private int objNumber = 0;

  private static int objCntA = 0;
  private static int objCntB = 0;
  private static int objCntC = 0;
```

Listing 4 - Constructor for class A.

```
  A(int val){//top-level class constructor
    aVar = val;
    objCntA++;//Increment object counter
    //Record the number of the object being
    // instantiated
    objNumber = objCntA;
    System.out.println(
      "In xstr for A, objCntA = " + objCntA);
  }//end constructor
```

Listing 5 - Beginning of the member class named B.

```
class B{//member class of A
  private int bVar;
  private int objNumber = 0;
```

Listing 6 - Constructor for class B.

```
    B(int val){//constructor
      bVar = val;
      //Increment static variable in top-level
      // class named A
      A.objCntB++;
      //Record the number of the object being
      // instantiated
      objNumber = objCntB;
      System.out.println(
          "In xstr for B, objCntB = " +
objCntB);
    }//end constructor
```

Listing 7 - Beginning of the member class named C.

Listing 7 - Beginning of the member class named C.

```
    class C extends X{//member class of B
      private int cVar;
      private A refToA;
      private B refToB;
      private String className = "C";
      private int objNumber = 0;
```

Listing 8 - Beginning of constructor for class C.

```
      C(int val){//constructor
        cVar = val;
        //Increment the object counter in the
        // top-level class named A.
        A.objCntC++;
        objNumber = A.objCntC;
        System.out.println(
                    "In xstr for C, objCntC =
"
                                      +
A.objCntC);
```

Listing 9 - The private member class named D.

```
      private class D{//member class of C
        D(){//constructor
          System.out.println(
            "Construct obj of private class
D.");
          System.out.println(
                    "  Private class file
name: "
                    +
this.getClass().getName());
        }//end constructor
      }//end class D
```

Listing 10 - More constructor code for class C.

```
        new D();
```

Listing 11 - More constructor code for class C.

Listing 11 - More constructor code for class C.

```
        refToA = new A(10);
        refToB = new B(20);

      }//end constructor
```

Listing 12 - The method named bShow.

```
    private void bShow(){
      System.out.println(
                      "In bShow, bVar = " +
bVar);
      System.out.println(
           "In bShow, objNumber = " +
objNumber);
      aShow();
    }//end bShow
```

Listing 13 - The method named aShow.

Listing 13 - The method named aShow.

```
  private void aShow(){
    System.out.println(
                      "In aShow, aVar = " +
aVar);
    System.out.println(
            "In aShow, objNumber = " +
objNumber);
  }//end aShow
```

Listing 14 - Beginning of the cShow method.

```
      public void cShow(){
        System.out.println("-1-");//separator
        System.out.println(
           "In cShow, objNumber = " +
objNumber);
        System.out.println(
                      "In cShow, cVar = " +
cVar);
```

Listing 15 - More cShow method code.

```
        System.out.println("-2-");//separator
        bShow();
```

Listing 16 - Call the aShow method.

```
        System.out.println("-3-");//separator
        aShow();
```

Listing 17 - More cShow method code.

Listing 17 - More cShow method code.

```
        System.out.println("-4-");//separator


System.out.println(getClass().getName());
        System.out.println(

this.getClass().getName());
        System.out.println(

C.this.getClass().getName());
        System.out.println(

B.C.this.getClass().getName());
        System.out.println(

A.B.C.this.getClass().getName());
```

Listing 18 - More cShow method code.

Listing 18 - More cShow method code.

```
        System.out.println("-5-");//separator

        System.out.println(

B.this.getClass().getName());
        System.out.println(

A.B.this.getClass().getName());
```

Listing 19 - More cShow method code.

```
        System.out.println("-6-");//separator

        System.out.println(

A.this.getClass().getName());
```

Listing 20 - More cShow method code.

Listing 20 - More cShow method code.

```
        System.out.println("-7-");

        System.out.println(
              "In cShow, bVar = " +
refToB.bVar);
        System.out.println(

refToB.getClass().getName());

        System.out.println(
              "In cShow, aVar = " +
refToA.aVar);
        System.out.println(

refToA.getClass().getName());
```

Listing 21 - More cShow method code.

```
        System.out.println("-8-");

        refToB.bShow();
```

Listing 22 - More cShow method code.

```
        System.out.println("-9-");

        refToA.aShow();
```

Listing 23 - Overridden toString method in class C.

```
    public String toString(){
      return "toString in Class C";
    }//end overridden toString
```

Listing 24 - More cShow method code.

Listing 24 - More cShow method code.

```
        System.out.println("-10-");
        System.out.println(
                     "className = " +
 className);
        System.out.println(toString());

        System.out.println(
               "className = " +
 super.className);
        System.out.println(super.toString());
      }//end cShow method
```

Note:

Listing 25 - Complete program listing.

```
/*File InnerClasses06.java
Copyright 2003 R.G.Baldwin

Rev 6/22/03

Illustrates the use of member classes.  Class B
is a member class of class A.  Class C is a
member class of class B.  Class D is a private
member class of class C.  An object is
instantiated from the class named A, which makes
it possible to instantiate an object of the
member class named B.  According to Flanagan,
this causes the object of the Class B to be
internally associated with the object of the
class named A.  The object of the class named B
is used to instantiate an object of the member
```

```
class named C.  This object is internally
associated with the object of the class B.

When the object of the class C is instantiated,
the constructor for that class instantiates
separate objects of the classes named A and B and
also instantiates an object of the private member
class named D.  The new and separate object of
the class B continues to be internally associated
with the original object of the Class A.

A variety of operations are performed from within
methods belonging to the object of the Class C to
illustrate the various characteristics of objects
instantiated from member classes.  Comments in
the code explain the purpose of each of those
operations.

The compilation of this program produces the
following class files:

A$B$C$D.class
A$B$C.class
A$B.class
A.class
InnerClasses06.class
X.class

The output from this program is shown below:

In xstr for A, objCntA = 1
In xstr for B, objCntB = 1
In xstr for C, objCntC = 1
Construct obj of private class D.
  Private class file name: A$B$C$D
In xstr for A, objCntA = 2
In xstr for B, objCntB = 2
```

```
-1-
In cShow, objNumber = 1
In cShow, cVar = 3
-2-
In bShow, bVar = 2
In bShow, objNumber = 1
In aShow, aVar = 1
In aShow, objNumber = 1
-3-
In aShow, aVar = 1
In aShow, objNumber = 1
-4-
A$B$C
A$B$C
A$B$C
A$B$C
A$B$C
-5-
A$B
A$B
-6-
A
-7-
In cShow, bVar = 20
A$B
In cShow, aVar = 10
A
-8-
In bShow, bVar = 20
In bShow, objNumber = 2
In aShow, aVar = 1
In aShow, objNumber = 1
-9-
In aShow, aVar = 10
In aShow, objNumber = 2
-10-
className = C
```

```
toString in Class C
className = X
toString in Class X


Tested using SDK 1.4.1 under WinXP
**********************************************/

public class InnerClasses06{
  public static void main(String[] args){
    //Instantiate an object of the member class
    // named C.  Note that it is necessary to
    // instantiate objects of the enclosing
    // classes as well.  Then invoke the public
    // method named cShow on the object of the
    // class named C.
    new A(1).new B(2).new C(3).cShow();
  }//end main
}//end class InnerClasses06
//===========================================//

//This class will be extended by the class named
// C, which is a member of the class named B,
// which is a member of the class named A.  This
// will illustrate that the inheritance
// hierarchy is independent of the containment
// hierarchy.
class X{//extends Object by default
  protected String className = "X";

  //Override the toString method
  public String toString(){
    return "toString in Class X";
  }//end overridden toString
}//end class X
//===========================================//
```

```
class A{
  private int aVar;
  private static int objCntA = 0;
  //Cannot place static variable in inner class.
  // Place it here instead.
  private static int objCntB = 0;
  //Cannot place static variable in inner class.
  // Place it here instead.
  private static int objCntC = 0;
  private int objNumber = 0;

  A(int val){//top-level class constructor
    aVar = val;
    objCntA++;//Increment object counter
    //Record the number of the object being
    // instantiated
    objNumber = objCntA;
    System.out.println(
          "In xstr for A, objCntA = " + objCntA);
  }//end constructor
  //-----------------------------------------//

  private void aShow(){
    System.out.println(
                     "In aShow, aVar = " + aVar);
    System.out.println(
           "In aShow, objNumber = " + objNumber);
  }//end aShow
  //=========================================//

  //Note that this class is defined internal to
  // the class named A.
  class B{//member class of A
    private int bVar;
    private int objNumber = 0;

    B(int val){//constructor
```

```
      bVar = val;
      //Increment static variable in top-level
      // class
      A.objCntB++;
      //Record the number of the object being
      // instantiated
      objNumber = objCntB;
      System.out.println(
          "In xstr for B, objCntB = " + objCntB);
    }//end constructor
    //-----------------------------------------//

    private void bShow(){
      System.out.println(
                    "In bShow, bVar = " + bVar);
      System.out.println(
           "In bShow, objNumber = " + objNumber);
      //Invoke the private method named aShow
      // belonging to the internally associated
      // object of the class named A.
      aShow();
    }//end bShow
    //=========================================//

    //Note that this class is defined internal to
    // the class named B.
    class C extends X{//member class of B
      private int cVar;
      private A refToA;
      private B refToB;
      private String className = "C";
      private int objNumber = 0;

      C(int val){//constructor

        cVar = val;
        //Instantiate separate objects of the
```

```
      // enclosing classes B and C.  Will
      // display info about them later.  The
      // object of the Class B is internally
      // associated with the original object of
      // the Class A.

      //Increment the object counter in the
      // top-level class.
      A.objCntC++;
      objNumber = A.objCntC;
      System.out.println(
                   "In xstr for C, objCntC = "
                                + A.objCntC);

      //Instantiate object of private member
      // class named D.
      new D();

      //Instantiate objects of enclosing
      // classes named A and B.

      refToA = new A(10);
      refToB = new B(20);

    }//end constructor

    public void cShow(){
      System.out.println("-1-");//separator
      //Display private member variables
      // belonging to this object.
      System.out.println(
         "In cShow, objNumber = " + objNumber);
      System.out.println(
                  "In cShow, cVar = " + cVar);
      System.out.println("-2-");//separator
      //Invoke the private method named bShow
      // in the internally associated object of
```

```
        // the class named B.  This method will,
        // in turn invoke the private method
        // named aShow in the object of the class
        // named A to which the object of the
        // class named B is internally
        // associated.
        bShow();
        System.out.println("-3-");//separator

        //Invoke the private method named aShow
        // in the internally associated object
        // of the class named A.
        aShow();
        System.out.println("-4-");//separator

        //Illustrate the syntax required to gain
        // access to the objects instantiated
        // from the classes named C, B, and A.
        // The first five statements produce the
        // same result.  The class names that are
        // displayed match the names of the class
        // files produced by the compilation
        // process.
        System.out.println(getClass().getName());
        System.out.println(
                     this.getClass().getName());
        System.out.println(
                    C.this.getClass().getName());
        System.out.println(
                  B.C.this.getClass().getName());
        System.out.println(
                 A.B.C.this.getClass().getName());
        System.out.println("-5-");//separator

        //The following two statements produce
        // the same output
        System.out.println(
```

```
                B.this.getClass().getName());
        System.out.println(
                  A.B.this.getClass().getName());
        System.out.println("-6-");//separator


        System.out.println(
                A.this.getClass().getName());
        System.out.println("-7-");

        //Display private instance variables and
        // class names belonging to separate
        // objects instantiated from the
        // enclosing classes named A and B.
        System.out.println(
              "In cShow, bVar = " + refToB.bVar);
        System.out.println(
                refToB.getClass().getName());

        System.out.println(
              "In cShow, aVar = " + refToA.aVar);
        System.out.println(
                refToA.getClass().getName());
        System.out.println("-8-");

        //Invoke the private bShow method on the
        // separate object instantiated from the
        // class named B in order to show the
        // object to which that object is
        // internally associated.  Even though
        // this object was instantiated from
        // within the constructor for the class
        // named C, it is internally associated
        // with the object of the class A that
        // was originally used to instantiate the
        // object of the class named C.
        refToB.bShow();
```

```
    System.out.println("-9-");

    //Invoke the private aShow method on the
    // separate object instantiated from the
    // class named A.
    refToA.aShow();
    System.out.println("-10-");

    //Illustrate the inheritance hierarchy to
    // which the class named C belongs by
    // getting and displaying the variable
    // named className from both the C class
    // and the X class.  Note that the
    // variable is protected in the X class.
    // Also invoke the overridden toString
    // methods belonging to the object
    // instantiated from the class named C.
    // One version is overridden in the class
    // named X and the other version is
    // overridden in the class named C.  Note
    // that the inheritance hierarchy is
    // totally independent of the containment
    // hierarchy.
    System.out.println(
                  "className = " + className);
    System.out.println(toString());
    //Note: cannot access private variable in
    // superclass named X. Must be protected,
    // package, or public.
    System.out.println(
           "className = " + super.className);
    System.out.println(super.toString());
  }//end cShow
  //-----------------------------------------//

  //Override the toString method
  public String toString(){
```

```
        return "toString in Class C";
      }//end overridden toString
      //=======================================//

      private class D{//member class of C
        D(){//constructor
          System.out.println(
            "Construct obj of private class D.");
          System.out.println(
                    "  Private class file name: "
                    + this.getClass().getName());
        }//end constructor
      //Note that all four class definitions end
      // here in the proper nested order.
      }//end class D
    }//end class C
  }//end class B
}//end class A
```

-end-

Java1638 Java OOP Local Classes
Baldwin explains the various relationships that exist among local classes and their enclosing classes.

Revised: Wed May 11 15:29:28 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
    - What can you include in a class definition?
    - What is a local class?
    - Why use local classes?
    - Local classes versus member classes
    - The scope of a local class
    - Restrictions on local classes
- Discussion and sample code
    - Class hierarchy
    - Overall program structure and behavior
        - The controlling class
        - The class named X
        - The class named Y
        - The class named A

        - The local class named B

- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing
- Figures
- Listings

Preface

This module is one in a collection of modules designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

> *(Editor's note: As you read this module, you will see that it was originally written around 2003. However, despite many improvements in Java since then, most of what was true then is still true in 2013.)*

This module makes several references to my website, which is located at http://www.dickbaldwin.com/toc.htm .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1 . Class files produced by the program.
- Figure 2 . Screen output.
- Figure 3 . Screen output.
- Figure 4 . Screen output.
- Figure 5 . Screen output.
- Figure 6 . Screen output.
- Figure 7 . Screen output.
- Figure 8 . Screen output.
- Figure 9 . Screen output.

Listings

- Listing 1 . The controlling class named InnerClasses07.
- Listing 2 . The class named X.
- Listing 3 . The class named Y.
- Listing 4 . Beginning of the class named A.
- Listing 5 . The showA method in class A.
- Listing 6 . The delay method in class A.
- Listing 7 . Beginning the meth method in class A.
- Listing 8 . Set value of blank final variable.
- Listing 9 . Beginning of class definition for local class B.
- Listing 10 . Instantiate two objects of local class B.
- Listing 11 . Call showB on the first object.
- Listing 12 . Beginning of showB method in local class B.
- Listing 13 . More code from showB method.
- Listing 14 . Use of the keyword this.
- Listing 15 . Illustrate the inheritance hierarchy.
- Listing 16 . Call showB method on second object.
- Listing 17 . Complete program listing.

Preview

What can you include in a class definition ?

There are several different kinds of items that can be included in a class definition. As you learned in the earlier modules in this series, the list includes:

- Static variables
- Instance variables
- Static methods
- Instance methods
- Constructors
- Static initializer blocks
- Instance initializers

Can also contain other class definitions

As you learned in the previous module , a class definition can also contain the following three kinds of *inner classes:*

- Member classes
- Local classes
- Anonymous classes

The previous module explained member classes. This module will explain local classes. The next module will explain anonymous classes.

What is a local class ?

A *local class* is a class that is defined within a block of Java code. While local classes are probably most frequently defined within methods and constructors, they can also be defined inside static initializer blocks and instance initializers.

As is the case for an object of a member class *(discussed in the previous module)*, an object of a local class must be internally linked to an object of the enclosing class. I will often refer to that object as the containing object, and make comments about the containment hierarchy.

A local class is truly an inner class, because an object of the local class cannot exist in the absence of an object of the enclosing class.

What about a local interface?

Interfaces defined within classes are implicitly static. This means that they are always *top-level.* There is no such thing as a member interface, a local interface, or an anonymous interface.

Why use local classes ?

Objects instantiated from local classes share many of the characteristics of objects instantiated from member classes. However, in some cases, a local class can be defined closer to its point of use than would be possible with a member class, leading to improved code readability.

Probably the most important benefit of local classes has to do with accessing the members of enclosing classes. Just like with member classes, methods of a local class have direct access to all the members of the enclosing classes, including private members. Thus the use of local classes can sometimes eliminate the requirement to connect objects together via constructor parameters.

> *(We will also see in the example program in this module that methods of a local class have direct access to protected members of the superclass of the enclosing class.)*

Can be particularly useful when ...

A local class can be particularly useful in those cases where

- There is no reason for an object of the local class to exist in the absence of an object of the enclosing class

- There is no reason for an object of the local class to exist outside a method of the enclosing class
- Methods of the object of the local class need access to members of the object of the enclosing class
- Methods of the object of the local class need access to **final** local variables and method parameters belonging to the method in which the local class is defined

This module approaches local classes from a somewhat theoretical viewpoint. The next module will approach local classes from a more practical viewpoint, including a comparison between local classes and anonymous classes.

Local classes versus member classes

A local class has approximately the same relationship to a member class that a local variable in a method has to an instance variable of the class containing the method.

The scope of a local class

As is the case with local variables, the name of a local class is visible and usable only within the block of code in which it is defined *(and blocks nested within that block).* Further, the name of the local class is visible and usable only to code following the class definition within that block.

The methods of a local class can use any ***final*** local variables or method parameters that are visible from the scope in which the local class is defined.

Similar to member classes

As mentioned earlier, local classes have many characteristics in common with member classes. This includes access to private fields and methods in the containing class. The thing that separates local classes from member

classes is the fact that local classes have access to local variables in the scope in which the local class is defined.

A big restriction

There is a big restriction, however, on the ability of methods of the local class to access local variables and method parameters. The methods in a local class can access local variables and method parameters only if they are declared **final** .

What does Flanagan have to say?

According to one of my favorite authors, David Flanagan, author of Java in a Nutshell, the methods in a local class don't really have access to local variables and method parameters. Rather, when an object of the local class is instantiated, copies of the **final** local variables and method parameters referred to by the object's methods are stored as instance variables in the object. The methods in the object of the local class really access those hidden instance variables. *(See the later section entitled Smoke and mirrors.)*

Thus, the local variables and method parameters accessed by the methods of the local class must be declared **final** to prevent their values from changing after the object is instantiated.

Restrictions on local classes

As with member classes, local classes cannot contain **static** members.

As with local variables, local classes cannot be declared **public** , **protected** , **private** , or **static** .

A local class cannot have the same name as any of its enclosing classes.

Smoke and mirrors

As I mentioned in the previous module, every class definition in a Java program, including nested top-level classes, member classes, local classes, and anonymous classes, produces a class file when the program is compiled. According to Flanagan,

> *"The Java Virtual Machine knows nothing about nested top-level classes and interfaces or the various types of inner classes. Therefore, the Java compiler must convert these new types into standard non-nested class files that the Java interpreter can understand. This is done through source code transformations that insert $ characters into nested class names. These source code transformations may also insert hidden fields, methods, and constructor arguments into the affected classes."*

A reference to the containing object

For example, the compiler automatically inserts a private instance variable in the local class to hold a reference to the containing object. It also inserts a hidden argument in all constructors for the local class, and passes the containing object's reference to the constructor for the local class. The modified constructor saves that reference in the private instance variable of the object of the local class. Thus each object instantiated from the local class contains a private reference to the containing object.

Accessing private members

In those cases where it is necessary for an object of the local class to access private members of the containing object, the compiler automatically creates and uses accessor methods that make such access possible.

Similar to your code

The bottom line is that the code that is automatically produced by the compiler is probably very similar to code that you would write if you were writing the program using only top-level classes. The good news is that you

don't have to write that extra code, and you don't have to maintain it. The extra code is written for you, and if you modify your class structure, the extra code is automatically modified accordingly.

Discussion and sample code

The paragraphs that follow will explain a program named **InnerClasses07** . This program is designed specifically to illustrate various characteristics of local classes. I will discuss the program in fragments. A complete listing is shown in Listing 17 near the end of the module.

This program illustrates the use of local classes. The program consists of a total of five classes:

- Four top-level classes named
    - InnerClasses07
    - A
    - X
    - Y
- A local class named B.

When compiled, the program produces the class files shown in Figure 1 . The file named A1B.class represents the local class. The remaining files in Figure 1 represent the four top-level classes.

Class hierarchy

In the previous module, I explained that once you understand the class file naming convention, you can determine from the file names how top-level classes and member classes are structured. However, the situation isn't nearly so clear when it comes to local classes and anonymous classes. This will become more apparent in the next module, which combines local classes and anonymous classes.

Overall program structure and behavior

Containment hierarchy

The program named **InnerClasses07** defines a local class named **B** inside an instance method named **meth** .

The method named **meth** is an instance method of a top-level class named **A** . The method named **meth** instantiates two separate objects of the local class named **B** , and calls a method named **showB** on each of them.

The method named **showB** displays certain data values that illustrate the characteristics of local classes, as well as the containment hierarchy among objects of the local class and an object of the containing class.

In this case, objects of the class named **B** are *contained within* an object of the class named **A** . *(The class named **A** is an enclosing class of the class named **B** .)*

Inheritance hierarchy

The top-level class named **A** extends the top-level class named **X** , *(which in turn, extends **Object**).*

The class named **B** is contained in or enclosed by the top-level class named **A** , but extends the top-level class named **Y** , *(which in turn, extends **Object**).*

There is no inheritance relationship between the classes **X** and **Y** *(aside from their common superclass named **Object**).*

There is no inheritance relationship between the classes **A** and **B** .

The method named **showB** also displays data values that are intended to demonstrate that the inheritance hierarchy is independent of the containment hierarchy.

> *(Note that while the containment hierarchy of local classes is independent of the inheritance hierarchy, it is technically possible to establish an inheritance relationship between a local class and one of its enclosing classes. For example, by making a couple of minor modifications, it is possible to cause the local class **B** in this program to extend the enclosing class **A** instead of the top-level class **Y** .)*

The controlling class

The entire controlling class named **InnerClasses07** , including the **main** method and a static variable named **baseTime** , is shown in Listing 1 .

The static variable named **baseTime** is initialized with the current date and time in milliseconds *(relative to January 1, 1970).* This static variable is used later as a base to establish the relative points in time that certain activities occur during the execution of the program.

The main method

The **main** method shown in Listing 1 instantiates a new object of the class named **A** and calls the method named **meth** on that object.

This method will sequentially instantiate two separate objects of a local class named **B** that is defined inside the method named **meth** . Then the method named **meth** will call a method named **showB** on each of those objects, causing them to display data values that illustrate the characteristics of local classes.

The class named X

The top-level class named **X** is shown in Listing 2 .

The class named **A** extends this class to illustrate the difference between the inheritance hierarchy and the containment hierarchy. Note that this class defines and initializes a protected instance variable, which will be accessed later to illustrate the inheritance hierarchy.

> *(Somewhat surprisingly, you will also see later that this protected instance variable belonging to the superclass of the enclosing class **A** is also accessible by methods belonging to an object of the local class via the containment hierarchy.)*

The class named Y

The top-level class named **Y** is shown in Listing 3 .

The local class named **B** extends this class to illustrate the difference between the inheritance hierarchy and the containment hierarchy.

Note that this class defines and initializes a protected instance variable. It also overrides the **toString** method, which is inherited from the **Object** class.

The instance variable and the overridden **toString** method will be accessed later to illustrate the inheritance hierarchy.

The class named A

The code in Listing 4 shows the beginning of the top-level class named **A** .

This code declares and initializes a private instance variable named **aTime** , which establishes the relative time that an object of class **A** is instantiated.

This private instance variable will be accessed directly by code belonging to an object of the local class **B** , which is contained by an object of the class **A** .

Listing 4 also shows the constructor for class **A** , which displays the time whenever an object is instantiated from class **A** .

The screen output

The code shown earlier in the **main** method of Listing 1 instantiates an object of class **A** . This causes the screen output shown in Figure 2 .

> *(Note that the relative time output value on your system may be different, depending on the speed of your system and the impact of other applications that may be running concurrently.)*

The private showA method

Listing 5 shows a private method named **showA** . This method displays the following information about an object instantiated from class **A** :

- The relative time that the object was instantiated.
- The value of the instance variable named **xVar** , which is an instance variable of the superclass of class **A** .
- The name of the class file that represents the class named **A** .

Code in methods belonging to an object of the local class named **B** , contained in an object of class **A** , has direct access to private members of objects of the containing or enclosing class.

The private **showA** method will be called by a method named **showB** , belonging to an object of the local class **B** , to demonstrate the truth of this statement.

The method named delay

Listing 6 shows a utility method named **delay** , which is an instance method of class **A** .

The purpose of the **delay** method is simply to insert a 30-millisecond delay in the execution of the program. This method is used to cause certain activities in the execution of the program to be spread far enough apart in time to make them distinguishable, one from the other, on the basis of relative time.

The method named meth

The method named **meth** is an instance method of the class **A** . The beginning of this method is shown in Listing 7 .

The method named **meth** contains a local class definition for a class named **B** , which we will examine later.

As you can see in Listing 7 , it also declares a **final** local variable named **methTime** . As you will see later, the method named **meth** instantiates two separate objects of local class **B** and calls a method named **showB** on each of those objects.

The method named **showB** displays various data values that illustrate the characteristics of local classes, including the value of the **final** local variable named **methTime** .

Access to final local variables

One of the characteristics of a local class is that objects of a local class have access to local variables within the same scope, provided that those local variables are declared **final** .

Blank final variables

The code in Listing 7 declares a **final** local variable named **methTime** . Because of the syntax used, this variable is known as a *blank* **final** variable. A blank final variable is not initialized when it is declared.

As with all local variables, the variable cannot be used until a value has been assigned to it. Because this variable is declared **final** , once a value has

been assigned to it, the value cannot be changed throughout the remaining life of the variable.

Set value of blank final variable

The code in Listing 8

- Inserts a delay
- Sets a value for the **final** local variable named **methTime**
- Displays the value that was set in **methTime**
- Inserts an additional delay before continuing

The value that is set in the variable named **methTime** is the relative time that the statement is executed.

The screen output

The code in the **main** method of Listing 1 calls **meth** on a new object of class **A** , producing the screen output shown in Figure 3 .

> *(Once again, the actual time value displayed by your system may be different, depending on the speed of your system and other factors as well.)*

The local class named B

The next block of code in the method named **meth** is the definition of a local class named **B** . The beginning of the class definition for local class **B** is shown in Listing 9 .

As with local variables, the class definition for a local class must appear before the code that attempts to instantiate the class. Code in the method named **meth** following the local class definition will instantiate and exercise objects of class **B** .

Local class **B** extends top-level class **Y** . This was done to illustrate that the inheritance hierarchy is independent of the containment hierarchy.

The constructor

The code in Listing 9 declares a private instance variable named **bTime** , which is used to store the relative time that an object of class **B** is constructed.

Listing 9 also shows the constructor for local class **B** , which establishes, saves, and then displays the relative time that that an object is instantiated.

> *(I will show you the screen output produced by this constructor shortly as I discuss code that instantiates objects of this class.)*

Instantiate two objects of class B

At this point, I am going to temporarily set aside the discussion of local class **B** and discuss code in the method named **meth** that immediately follows the definition of local class **B** . This code is shown in Listing 10 .

Listing 10 shows the beginning of code that is called when the method named **meth** is called. This code begins by instantiating two objects from the class named **B** , with a delay inserted between the instantiation of the two objects.

The screen output

The code in Listing 10 , in conjunction with the constructor code in Listing 9 produces the screen output shown in Figure 4 as each of the two objects of local class **B** are instantiated.

Call showB on the first object

Following this, the code in the method named **meth** calls the method named **showB** on each of the two objects. The method named **showB** will, in turn,

call the method named **showA** on the containing object instantiated from the class named **A** . The third line of code in Listing 11 calls the method named **showB** on the first object instantiated in Listing 10 .

The method named showB

That brings us back to a discussion of the method named **showB** , which is an instance method of local class **B** . The beginning of the **showB** method is shown in Listing 12 . This method displays several private and protected variables, some of which belong to the containing object instantiated from the top-level class named **A** .

> *(Note that code in this method has direct access to **xVar** , which is a protected member variable of the superclass of the class named **A** .)*

Items that are displayed

The code in Listing 12 displays

- A visual line separator, -1-
- A private instance variable, **bTime** , belonging to the object instantiated from local class **B**
- A private instance variable, **aTime** , belonging to the containing object instantiated from the top-level class **A**
- A **final** local variable, **methTime** , belonging to the method named **meth** , in which the local class **B** is defined
- A protected instance variable, **xVar** , of the superclass of the class **A** , from which the containing object was instantiated.
- A protected instance variable, **yVar** , of the superclass of the local class **B**
- The name of the class file that represents the local class **B**

The screen output

The combined code in Listing 11 and Listing 12 produces the output shown in Figure 5 for the first object instantiated from local class **B** .

Call private method showA

One of the important characteristics of local classes is that the methods of objects instantiated from local classes have direct access to all of the members, including private members, of all the containing classes in the containment hierarchy.

> *(In this case, an object of the local class **B** has only one containing class, an object instantiated from the top-level class **A** .)*

Continuing with the method named **showB** , in local class **B** , the code in Listing 13 displays a line separator, -2-, and then calls the private method named **showA** on the containing object. This illustrates the containment hierarchy.

The screen output

The method named **showA** is shown in Listing 5 . The combination of the code in Listing 13 and Listing 5 produces the screen output shown in Figure 6 .

Figure 6 displays

- The relative time that the containing object of the class **A** was instantiated
- A protected instance variable in the superclass of the class from which the containing object was instantiated
- The name of the class file that represents the top-level class named **A**

Use of the keyword this

As is the case with member classes, *(discussed in the previous module)*, objects of local classes use a special syntax of the keyword ***this*** to gain access to objects in the containment hierarchy.

The code in Listing 14 shows how to use regular syntax to gain access to the current object, and how to use special syntax to gain access to the containing object.

Having gained access to the two objects, the code in Listing 14 gets and displays the values of private instance variables belonging to those objects.

The screen output

The code in Listing 14 produced the screen output shown in Figure 7 .

You saw these same two values displayed earlier in Figure 4 and Figure 2 .

Illustrate the inheritance hierarchy

Remember that the inheritance hierarchy is independent of the containment hierarchy. The code in Listing 15 illustrates the inheritance hierarchy by

- Calling the overridden **toString** method belonging to the local class **B**
- Using the **super** keyword to call the overridden **toString** method belonging to the class **Y** , which is the superclass of the local class **B**

Listing 15 also defines the overridden **toString** method belonging to local class **B** . The overridden **toString** method belonging to class **Y** is shown in Listing 3 .

The screen output

The combined code in Listing 15 and Listing 3 produces the screen output shown in Figure 8 . *(Once again, you have seen these same values displayed in earlier Figures.)*

That concludes the results of calling the **showB** method on the first object instantiated from local class **B** .

Call showB method on second object

The code in Listing 10 instantiated two objects from local class **B** , and saved the object's references in instance variables named **obj1** and **obj2** . The screen output that I have been discussing resulted from calling the **showB** method on **obj1** in Listing 11 .

Listing 16 calls the **showB** method on **obj2** . Listing 16 also signals the end of the method named **meth** , and the end of the class named **A** .

The screen output The code in Listing 16 produces the screen output shown in Figure 9

Important considerations The important things to note about the output shown in Listing 16 include:

- The second object instantiated from local class **B** is definitely a different object from the first object instantiated from local class **B** , as evidenced by a different value for **bTime** . *(Compare the value of **bTime** in Figure 9 with the value of **bTime** in Figure 5 .)* In other words, the second object was instantiated after the first object was instantiated.
- Even though the two objects instantiated from local class **B** are different objects, they both belong to the same containing object, as evidenced by the same values for **aTime** in Figure 9 and Figure 5 .
- The two objects instantiated from local class **B** each access the same **final** local variable belonging to the method named **meth** , as evidenced by the same values for **methTime** in Figure 9 and Figure 5 .

Run the program

I encourage you to copy the code from Listing 17 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In addition to a number of other items, a class definition can contain:

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

Member classes were explained in the previous module. This module explains local classes. The next module will explain anonymous classes.

A *local class* is a class that is defined within a block of Java code. While local classes are probably most frequently defined within method and constructors, they can also be defined inside static initializer blocks and instance initializers.

An object of the local class must be internally linked to an object of the enclosing class *(which I often refer to as the containing object).* A local class is truly an inner class because an object of the local class cannot exist in the absence of an object of the enclosing class.

The methods of a local class have direct access to all the members in the hierarchy of enclosing classes, including private members. In addition, the methods of local classes have access to **final** local variables and **final** method parameters in the scope in which the local class is defined.

The containment hierarchy of local classes is independent of the inheritance hierarchy. However, it is technically possible to establish an inheritance relationship between a local class and one of its enclosing classes.

Local classes may not be declared **public** , **protected** , **private** , or **static** .

Local classes cannot contain **static** members.

A local class has approximately the same relationship to a member class that a local variable in a method has to an instance variable of the class containing the method.

What's next?

The next module in this series will explain and discuss anonymous classes, and will also compare anonymous classes to local classes.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Local Classes
- File: Java1638.htm
- Published: 11/19/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program is provided in Listing 17 .

Figures

Figure 1 - Class files produced by the program.

```
A$1$B.class
A.class
InnerClasses07.class
X.class
Y.class
```

Figure 2 - Screen output.

```
In xstr for A, aTime = 10
```

Figure 3 - Screen output.

```
In meth, methTime = 40
```

Figure 4 - Screen output.

```
----------------------
Instantiate first B-object
In xstr for B, bTime = 70
Delay and instantiate second B-object
In xstr for B, bTime = 100
```

Figure 5 - Screen output.

Figure 5 - Screen output.

```
-----------------------
Display first B-Object
-1-
In showB, private bTime = 70
In showB, private aTime = 10
In showB, final methTime = 40
In showB, protected xVar = 1000
In showB, protected yVar = 2000
In showB, class name = A$1$B
```

Figure 6 - Screen output.

```
-2-
In showA, aTime = 10
In showA, xVar = 1000
In showA, class name = A
```

Figure 7 - Screen output.

Figure 7 - Screen output.

```
-3-
In showB, bTime = 70
In showB, aTime = 10
```

Figure 8 - Screen output.

```
-4-
toString in class B, bTime = 70
toString in class Y, yVar = 2000
```

Figure 9 - Screen output.

Figure 9 - Screen output.

```
----------------------
Display second B-Object
-1-
In showB, private bTime = 100
In showB, private aTime = 10
In showB, final methTime = 40
In showB, protected xVar = 1000
In showB, protected yVar = 2000
In showB, class name = A$1$B
-2-
In showA, aTime = 10
In showA, xVar = 1000
In showA, class name = A
-3-
In showB, bTime = 100
In showB, aTime = 10
-4-
toString in class B, bTime = 100
toString in class Y, yVar = 2000
```

Listings

Listing 1 - The controlling class named InnerClasses07.

Listing 1 - The controlling class named InnerClasses07.

```
public class InnerClasses07{

  static long baseTime = new Date().getTime();

  public static void main(String[] args){
    new A().meth();
  }//end main
}//end class InnerClasses07
```

Listing 2 - The class named X.

```
class X{
  protected int xVar = 1000;
}//end class X
```

Listing 3 - The class named Y.

Listing 3 - The class named Y.

```
class Y{
  protected int yVar = 2000;

  public String toString(){
    return "toString in class Y, yVar = " +
yVar;
  }//end overridden toString
}//end class Y
```

Listing 4 - Beginning of the class named A.

```
class A extends X{
  private long aTime = new Date().getTime() -

InnerClasses07.baseTime;

  A(){//constructor
    System.out.println(
              "In xstr for A, aTime = " +
aTime);
  }//end constructor
```

Listing 5 - The showA method in class A.

```
  private void showA(){
    System.out.println(
                    "In showA, aTime = " +
aTime);
    System.out.println(
                      "In showA, xVar = " +
xVar);
    System.out.println("In showA, class name =
"
                              +
getClass().getName());
  }//end showA
```

Listing 6 - The delay method in class A.

```
  void delay(){
    try{
      Thread.currentThread().sleep(30);
    }catch(InterruptedException e){
      System.out.println(e);
    }//end catch
  }//end delay
```

Listing 7 - Beginning the meth method in class A.

```
  void meth(){
    final long methTime;
```

Listing 8 - Set value of blank final variable.

```
    delay();
    methTime = new Date().getTime() -

InnerClasses07.baseTime;
    System.out.println(
              "In meth, methTime = " +
methTime);
    delay();
```

Listing 9 - Beginning of class definition for local class B.

Listing 9 - Beginning of class definition for local class B.

```
    class B extends Y{
      private long bTime;

      B(){//constructor
        bTime = new Date().getTime() -

InnerClasses07.baseTime;
        System.out.println(
              "In xstr for B, bTime = " +
bTime);
      }//end constructor
```

Listing 10 - Instantiate two objects of local class B.

```
    System.out.println("----------------------
");
    System.out.println(
                   "Instantiate first B-
object");
    final B obj1 = new B();
    System.out.println(
        "Delay and instantiate second B-
object");
    delay();
    final B obj2 = new B();
```

Listing 11 - Call showB on the first object.

```
    System.out.println("----------------------
");
    System.out.println("Display first B-
Object");
    obj1.showB()
```

Listing 12 - Beginning of showB method in local class B.

Listing 12 - Beginning of showB method in local class B.

```
//Continuing with local class B definition
      void showB(){
        System.out.println("-1-");
        System.out.println(
           "In showB, private bTime = " +
bTime);
        System.out.println(
           "In showB, private aTime = " +
aTime);
        System.out.println(
                  "In showB, final methTime =
" +

methTime);
        System.out.println(
           "In showB, protected xVar = " +
xVar);
        System.out.println(
           "In showB, protected yVar = " +
yVar);
        System.out.println(
                     "In showB, class name =
" +

getClass().getName());
```

Listing 13 - More code from showB method.

Listing 13 - More code from showB method.

```
        System.out.println("-2-");
        showA();
```

Listing 14 - Use of the keyword this.

```
        System.out.println("-3-");
        System.out.println(
              "In showB, bTime = " +
this.bTime);
        System.out.println(
            "In showB, aTime = " +
A.this.aTime)
```

Listing 15 - Illustrate the inheritance hierarchy.

Listing 15 - Illustrate the inheritance hierarchy.

```
        System.out.println("-4-");
        System.out.println(toString());
        System.out.println(super.toString());
      }//end showB
      //-------------------------------------
-//

      //Overridden toString method
      public String toString(){
        return
         "toString in class B, bTime = " +
bTime;
      }//end overridden toString
    }//end local class B
```

Listing 16 - Call showB method on second object.

Listing 16 - Call showB method on second object.

```
    System.out.println("----------------------
");
    System.out.println(
                      "Display second B-
Object");
    obj2.showB();

  }// end meth
}//end class A
```

Note:

Listing 17 - Complete program listing.

```
/*File InnerClasses07.java
Copyright 2003 R.G.Baldwin

Illustrates the use of local classes.

This program defines a local class named B inside
an instance method named meth.  The method named
meth is an instance method of a class named A.

The method named meth instantiates two separate
objects of the local class named B, and calls
on each of them a method named showB, which
displays certain data values that illustrate the
characteristics of local classes, a well as the
relationships among objects of a local class and
an object to which that object is internally
associated.  In this case, objects of the class
named B are internally associated with an object
```

```
of the class named A.

This program produces the following class files
when compiled:

A$1$B.class
A.class
InnerClasses07.class
X.class
Y.class

The file named A$1$B.class represents the local
class named B.

This program produces the following output:

In xstr for A, aTime = 10
In meth, methTime = 40
----------------------
Instantiate first B-object
In xstr for B, bTime = 70
Delay and instantiate second B-object
In xstr for B, bTime = 100
----------------------
Display first B-Object
-1-
In showB, private bTime = 70
In showB, private aTime = 10
In showB, final methTime = 40
In showB, protected xVar = 1000
In showB, protected yVar = 2000
In showB, class name = A$1$B
-2-
In showA, aTime = 10
In showA, xVar = 1000
In showA, class name = A
-3-
```

```
In showB, bTime = 70
In showB, aTime = 10
-4-
toString in class B, bTime = 70
toString in class Y, yVar = 2000
----------------------
Display second B-Object
-1-
In showB, private bTime = 100
In showB, private aTime = 10
In showB, final methTime = 40
In showB, protected xVar = 1000
In showB, protected yVar = 2000
In showB, class name = A$1$B
-2-
In showA, aTime = 10
In showA, xVar = 1000
In showA, class name = A
-3-
In showB, bTime = 100
In showB, aTime = 10
-4-
toString in class B, bTime = 100
toString in class Y, yVar = 2000


Tested using SDK 1.4.1 under WinXP
**********************************************/
import java.util.Date;

public class InnerClasses07{
  //The following static variable is used as a
  // base to establish the relative points in
  // time that certain activities occur during
  // the execution of the program.
  static long baseTime = new Date().getTime();
```

```
  public static void main(String[] args){
    //Instantiate a new object of the class named
    // A and call the method named meth on that
    // object.  This method will sequentially
    // instantiate two separate objects of a
    // local class named B that is defined inside
    // the method named meth.  Then it will
    // call a method named showB on each of
    // those objects to cause them to display
    // various data values that illustrate the
    // characteristics of local classes.
    new A().meth();
  }//end main
}//end class InnerClasses07
//===========================================//

//The class named A extends this class to
// illustrate the difference between the
// inheritance hierarchy and the containment
// hierarchy.
class X{
  protected int xVar = 1000;
}//end class X
//===========================================//

//The local class named B extends this class to
// illustrate the difference between the
// inheritance hierarchy and the containment
// hierarchy.
class Y{
  protected int yVar = 2000;

  //Overridden toString method
  public String toString(){
    return "toString in class Y, yVar = " + yVar;
  }//end overridden toString
}//end class Y
```

```
//=============================================//

class A extends X{
  //Establish the relative time that the object
  // of class A is instantiated.
  private long aTime = new Date().getTime() -
                         InnerClasses07.baseTime;

  A(){//constructor
    System.out.println(
              "In xstr for A, aTime = " + aTime);
  }//end constructor
  //---------------------------------------//

  //Displays information about the object
  // instantiated from class A.
  private void showA(){
    System.out.println(
                   "In showA, aTime = " + aTime);
    System.out.println(
                     "In showA, xVar = " + xVar);
    System.out.println("In showA, class name = "
                         + getClass().getName());
  }//end showA
  //---------------------------------------//

  //Method used to insert a time delay of 30
  // milliseconds.
  void delay(){
    try{
      Thread.currentThread().sleep(30);
    }catch(InterruptedException e){
      System.out.println(e);
    }//end catch
  }//end delay
  //---------------------------------------//
```

```
//This method contains a local class definition
// for a class named B.  The method
// instantiates two separate objects of class B
// and calls a method named showB on each of
// the objects.  The method named showB
// displays various data values that illustrate
// the characteristics of local classes.
void meth(){
  //The following local variable must be final
  // to be accessible from a local class.  This
  // is a blank final variable whose value can
  // be set once and never changed after that.
  final long methTime;

  //Delay and then set the value of the blank
  // final local variable to a relative time.
  // Then delay again before continuing.
  delay();
  methTime = new Date().getTime() -
                        InnerClasses07.baseTime;
  System.out.println(
            "In meth, methTime = " + methTime);
  delay();
//----------------------------------------//

  //This is the definition of a local class
  // named B.  Note that as with local
  // variables, the class definition must
  // appear before the code that attempts to
  // instantiate the class.  The class extends
  // class Y to illustrate that the inheritance
  // hierarchy is independent of the
  // containment hierarchy.
  class B extends Y{
    private long bTime;

    B(){//constructor
```

```
      //Establish the relative time that the
      // object is instantiated.
      bTime = new Date().getTime() -
                        InnerClasses07.baseTime;
      System.out.println(
            "In xstr for B, bTime = " + bTime);
    }//end constructor

    void showB(){
      //Display private and protected
      // variables, some of which belong to the
      // internally associated object
      // instantiated from the class named A.
      // Note that code in this method has
      // access to xVar, which is a protected
      // member variable of a superclass of the
      // class named A.
      System.out.println("-1-");
      System.out.println(
         "In showB, private bTime = " + bTime);
      System.out.println(
         "In showB, private aTime = " + aTime);
      System.out.println(
                "In showB, final methTime = " +
                                    methTime);
      System.out.println(
         "In showB, protected xVar = " + xVar);
      System.out.println(
         "In showB, protected yVar = " + yVar);
      System.out.println(
                    "In showB, class name = " +
                         getClass().getName());

      System.out.println("-2-");
      //Call the private method named showA
      // in the internally associated object
      // instantiated from the class named A,
```

```
      // to illustrate the containment
      // hierarchy.
      showA();

      System.out.println("-3-");
      //Show how to access individual objects,
      // including the internally associated
      // object instantiated from the class
      // named A.
      System.out.println(
            "In showB, bTime = " + this.bTime);
      System.out.println(
          "In showB, aTime = " + A.this.aTime);

      System.out.println("-4-");
      //Illustrate the inheritance hierarchy by
      // calling the overridden toString
      // methods belonging to the class named B
      // and its superclass named Y.
      System.out.println(toString());
      System.out.println(super.toString());
    }//end showB
    //---------------------------------------//

    //Overridden toString method
    public String toString(){
      return
       "toString in class B, bTime = " + bTime;
    }//end overridden toString
  }//end local class B

  //This is the code that is executed when this
  // method named meth is called.  Instantiate
  // two objects from the class named B and
  // call the method named showB on each of
  // them.  Those methods will, in turn call
  // the method named showA on the internally
```

```
    // associated object instantiated from the
    // class named A.  Insert a delay between the
    // instantiation of the two objects.
    System.out.println("----------------------");
    System.out.println(
                   "Instantiate first B-object");
    final B obj1 = new B();
    System.out.println(
        "Delay and instantiate second B-object");
    delay();
    final B obj2 = new B();

    System.out.println("----------------------");
    System.out.println("Display first B-Object");
    obj1.showB();
    System.out.println("----------------------");
    System.out.println(
                     "Display second B-Object");
    obj2.showB();

  }// end meth
}//end class A
//=============================================//
```

-end-

Java1640 Java OOP Anonymous Classes
Baldwin explains anonymous classes from a practical viewpoint, including a comparison between anonymous classes and local classes.

Revised: Wed May 11 15:54:45 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Preface
  - Purpose of this module
  - Viewing tip
    - Figures
    - Listings
- Preview
  - What can you include in a class definition?
  - What is an anonymous class?
  - What about an anonymous interface?
  - Why use anonymous classes?
  - Anonymous classes versus local classes
  - Restrictions on the use of anonymous classes
  - Smoke and mirrors
  - Syntax for anonymous classes
- Discussion and sample code
  - Class file names
  - Program structure and behavior
    - A local class

        - Three anonymous classes
        - The screen output
        - The controlling class
        - Local and anonymous classes inside GUI constructor
        - The GUI class
        - An anonymous inner class for low-level event handling
        - Instantiating and registering a MouseListener object
        - Extending the WindowAdapter class

    - The GUI remains on the screen until terminated
    - Simple event handlers

- Run the program
- Summary
- Miscellaneous
- Complete program listing
- Figures
- Listings

Preface

This module is one in a collection of modules designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

> *(Editor's note: As you read this module, you will see that it was originally written around 2003. However, despite many improvements in Java since then, most of what was true then is still true in 2013.)*

This module makes several references to my website, which is located at http://www.dickbaldwin.com/toc.htm .

Purpose of this module

This module explains anonymous classes from a practical viewpoint, including a comparison between anonymous classes and local classes.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1 . Syntax for anonymous classes.
- Figure 2 . Program GUI.
- Figure 3 . Class file names.
- Figure 4 . Screen output.
- Figure 5 . Screen output.
- Figure 6 . Screen output.
- Figure 7 . Screen output.
- Figure 8 . Screen output.
- Figure 9 . Screen output.

Listings

- Listing 1 . The controlling class.
- Listing 2 . Beginning of the GUI class.
- Listing 3 . Beginning of the BaldButton class.
- Listing 4 . The processMouseEvent method.
- Listing 5 . Beginning of anonymous inner class.
- Listing 6 . Overridden processMouseEvent method.
- Listing 7 . Register a MouseListener object.
- Listing 8 . Implementing the interface.

- Listing 9 . Registering a WindowListener on the Frame .
- Listing 10 . Complete program listing.

Preview

What can you include in a class definition ?

There are several different kinds of items that can be included in a class definition. As you learned in the earlier modules in this series, the list includes:

- Static variables
- Instance variables
- Static methods
- Instance methods
- Constructors
- Static initializer blocks
- Instance initializers

Can also contain other class definitions

As you also learned in previous modules, a class definition can also contain the following kinds of *inner classes:*

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

The previous two modules explained member classes and local classes. This module will explain anonymous classes.

What is an anonymous class ?

I'm going to begin my discussion with a quotation from one of my favorite authors, David Flanagan, author of *Java in a Nutshell* .

> *"An anonymous class is essentially a local class without a name."*

If you have read the previous module, you should know quite a lot about local classes at this point in time. Continuing with Flanagan's words,

> *"Instead of defining a local class and then instantiating it, you can often use an anonymous class to combine these two steps... an anonymous class is defined by a Java* ***expression*** *, not a Java* ***statement*** *. This means that an anonymous class definition can be included within a larger Java expression..."*

As you will see from the sample program in this module, anonymous class definitions are often included as arguments to method calls.

As is the case for an object of a member class or a local class *(discussed in previous modules)*, an object of an anonymous class must be internally linked to an object of the enclosing class.

Thus, an anonymous class is truly an inner class, because an object of the anonymous class cannot exist in the absence of an object of the enclosing class.

What about an anonymous interface ?

Interfaces defined within classes are implicitly static. This means that they are always *top-level.* There is no such thing as a member interface, a local interface, or an anonymous interface.

Why use anonymous classes ?

As with local classes, objects instantiated from anonymous classes share many of the characteristics of objects instantiated from member classes. However, in some cases, an anonymous class can be defined closer to its point of use than would be possible with a member class or a local class. Once you become accustomed to the somewhat cryptic syntax used with anonymous classes, this can often lead to improved code readability.

Probably the most important benefit of anonymous classes has to do with accessing the members of enclosing classes. Just like with member classes and local classes, methods of an anonymous class have direct access to all the members of the enclosing classes, including private members. Thus the use of anonymous classes can often eliminate the requirement to connect objects together via constructor parameters.

In addition, although not demonstrated in this module, as with local classes, objects of anonymous classes have access to **final** local variables that are declared within the scope of the anonymous class.

Can be particularly useful when ...

An anonymous class can be particularly useful in those cases where

- There is no reason for an object of the anonymous class to exist in the absence of an object of the enclosing class.
- There is no reason for an object of the anonymous class to exist outside a method of the enclosing class.
- Methods of the object of the anonymous class need access to members of the object of the enclosing class.
- Methods of the object of the anonymous class need access to **final** local variables and method parameters belonging to the method in which the anonymous class is defined.
- Only one instance of the anonymous class is needed.
- There is no need for the class to have a name that is accessible elsewhere in the program.

Anonymous classes versus local classes

Once again, according to David Flanagan,

> *"...an anonymous class behaves just like a local class, and is distinguished from a local class merely in the syntax used to define and instantiate it."*

Unlike a local class, however, an anonymous class cannot define a constructor. However, an anonymous class can define an instance initializer, which can provide some of the benefits of a constructor.

> *(I discussed instance initializers in detail in an earlier tutorial titled The Essence of OOP using Java, Instance Initializers . As you may recall, a primary shortcoming of an instance initializer as compared to a constructor is that an instance initializer cannot accept incoming parameters.)*

Restrictions on the use of anonymous classes

Because an anonymous class has no name, and the definition and instantiation of the class appear in a single expression, only one instance of each anonymous class can be created. If you need more than one instance of the class, you should probably use a local class, a member class, or a top-level class instead.

As mentioned above, it is not possible to define constructors for anonymous classes. If you need to use a constructor when you instantiate the class, you should probably use a local class, a member class, or a top-level class instead.

As with member classes and local classes, anonymous classes cannot contain **static** members.

As with local variables and local classes, anonymous classes cannot be declared **public** , **protected** , **private** , or **static** . In fact, no modifiers can be specified in the definition of an anonymous class.

Smoke and mirrors

As I told you in my earlier modules on local classes, the methods in an anonymous class don't really have access to local variables and method parameters. Rather, when an object of the anonymous class is instantiated, copies of the **final** local variables and method parameters referred to by the object's methods are stored as instance variables in the object. The methods in the object of the anonymous class really access those hidden instance variables.

Thus, the local variables and method parameters accessed by the methods of the local class must be declared **final** to prevent their values from changing after the object is instantiated.

There are some additional activities involving *smoke and mirrors* taking place behind the scenes when you define and instantiate an anonymous class. Generally speaking, this involves the automatic generation of code to cause things to behave as they do. The good news is that you don't have to write that extra code, and you don't have to maintain it. The extra code is written for you, and if you modify your class structure, the extra code is automatically modified accordingly.

You can read about the code that is automatically generated in my earlier modules on local classes and member classes.

Syntax for anonymous classes

Before getting into actual code in the sample program, I want to explain the syntax used to define and instantiate an anonymous class.

The definition and instantiation of an anonymous class uses one or the other of the two expressions shown in Figure 1 .

Usually, this expression is included inside a larger overall expression, such as an argument to a method call.

What does the first expression mean?

Here is how I usually explain this syntax to my students. The first expression in Figure 1 starts out fairly normal, but becomes cryptic very quickly.

This expression instantiates a new object from an unnamed and previously undefined class. That class automatically *extends* the class named **className** , and cannot explicitly implement any interfaces.

The body of the new class is given by **classBody** .

The result of executing this expression is that:

- a new class that extends **className** is defined,
- a new object of the new class is instantiated, and
- the expression is replaced by a reference to the new object.

Example usage

If this expression appears as the right operand of an assignment operator, the new object's reference is saved in the left operand of the assignment operator.

If the expression appears as an argument in a method call, the new object's reference is passed to the method.

If the expression appears in some other form of larger overall expression, the new object's reference is handed over to the surrounding expression to be used appropriately.

What about instantiating an interface?

The second expression in Figure 1 starts out very weird. To my knowledge, there is no other situation in Java where you apply the **new** operator to the name of an interface.

From the beginning, you have been told *(or should have been told)* that you cannot instantiate an object of an interface. *(An interface is implicitly abstract and it doesn't have a constructor, not even a default constructor.)*

However, you can instantiate an object of a class that implements none, one, or more interfaces.

The correct interpretation of the second expression in Figure 1 is as follows. This expression instantiates a new object from an unnamed and previously undefined class. That class automatically *implements* the interface named **interfaceName** , and it automatically *extends* the class named **Object** .

The class can explicitly implement one, and only one interface, and cannot extend any class other than **Object** .

Once again, the body of the new class is given by **classBody** .

As in the case of the first expression in Figure 1 , the result of executing this second expression is that

- a new class that implements **interfaceName** is defined,
- a new object of the new class is instantiated, and
- the expression is replaced by a reference to the new object.

That reference is handed over to the surrounding expression to be used appropriately.

What about constructor parameters?

As mentioned earlier in this module, since the new class doesn't have a name, it isn't possible to define a constructor for the new class.

According to Flanagan *(with regard to the first expression in Figure 1)* ,

> *"Any arguments you specify between the parentheses following the superclass name in an anonymous class definition are implicitly passed to the superclass constructor."*

Thus, it is possible to define an anonymous class that extends a class whose constructor requires parameters, and to pass those parameters to the superclass constructor when the anonymous class is instantiated.

The parentheses following **interfaceName** in the second expression in Figure 1 must always be empty. In this case, the superclass is always **Object** , which never expects constructor parameters.

Discussion and sample code

The paragraphs that follow will explain a program named **InnerClasses08** . This program is designed specifically to illustrate anonymous classes, and to compare anonymous classes with local classes.

I will discuss the program in fragments. A complete listing of the program is provided in Listing 10 .

When the program is executed, it produces the GUI shown in Figure 2 . I will refer back to this image during the discussion of the program.

Class file names

This program consists of a total of six classes:

- Two top-level classes
    - InnerClasses08.class
    - GUI.class
- One local class

    - GUI1BaldButton.class

- Three anonymous classes
    - GUI$1.class
    - GUI$2.class
    - GUI$3.class

When compiled, the program produces the class files shown in Figure 3 .

> *(As you can see, the anonymous classes are not truly anonymous, since the files that represent them must have names. Generally, however, the establishment of the individual names is beyond the control of the programmer, and the names are not known to the program in a way that makes it possible to refer to them by name.)*

Program structure and behavior

This program is designed to illustrate the use of local classes and anonymous classes in a very practical way. It illustrates one implementation of a local class and three different implementations of anonymous classes. The program compares the local class with an anonymous class designed to accomplish the same purpose. The program also illustrates the use of instance initializers as an alternative to constructors.

A local class

The program defines and uses a local class to instantiate an object to handle *mouse clicked* events on a button with ***low-level*** event handling. This class uses a constructor to enable mouse events on a new extended **Button** class. It also uses a constructor to display the name of the class file.

Three anonymous classes

An anonymous class to compare with the local class

The program also defines and uses an anonymous class to instantiate an object to handle *mouse clicked* events on a button with *low-level* event handling.

This class uses an instance initializer to enable mouse events on a new extended **Button** class. It also uses an instance initializer to display the name of the class file.

This class and the local class described above provide a direct comparison between the use of local classes and anonymous classes to serve the same purpose.

An anonymous class that implements an interface

The program illustrates the use of an anonymous class that implements the **MouseListener** interface, to instantiate an object to handle mouse clicked events using the source-listener event model *(sometimes referred to as the delegation event model or the JavaBeans event model).* The anonymous class uses an instance initializer to display the name of the class file.

An anonymous class that extends an existing class

The program illustrates the use of an anonymous class that extends the **WindowAdapter** class, to instantiate an object to handle window events fired by the close button in the upper-right corner of the **Frame** object shown in Figure 2 . This class also uses the source-listener event model, and uses an instance initializer to display the name of the class file.

The screen output

The program produces the screen output shown in Figure 4 when

- The program is started

- Each button shown in Figure 2 is clicked once in succession, going from left to right
- The close button in the upper-right corner of the **Frame** object in Figure 2 is clicked

When the close button is clicked, the program produces the last line of text in Figure 4 and terminates. I will identify the code that produces each line of output text in the discussion of the program that follows.

The controlling class

The controlling class for the program is shown in Listing 1 .

As you can see, the controlling class is very simple, with the **main** method instantiating an object of the **GUI** class. This results in the GUI that is pictured in Figure 2 .

Local and anonymous classes inside GUI constructor

The local class and the three anonymous classes are defined inside the constructor for the **GUI** class.

> *(Recall that local classes and anonymous classes are defined inside code blocks, which often place them inside methods and constructors, but you can also place them inside static initializer blocks and instance initializers.)*

The first four lines of the output text in Figure 4 are produced by constructors and instance initializers in the local and anonymous classes. Therefore, those four lines of text are produced when the new object of the **GUI** class is instantiated.

The GUI class

As is often the case, the **GUI** class used to create the visual GUI shown in Figure 2 consists solely of a constructor. Basically, this constructor

- places three buttons in the frame and
- registers event handlers on the buttons and on the frame.

Once the **GUI** object is constructed and appears on the screen, all further activity in the program occurs under control of the event handlers associated with the buttons and the frame.

> *(You can learn more about event handling here .)*

The GUI constructor

The **GUI** class, and the constructor for that class begin in Listing 2 .

As you can see, the **GUI** class extends **Frame** , so that an object of the class *is a* frame.

The constructor code shown in Listing 2 simply sets values for the *layout*, *size*, and *title* properties of the frame.

The BaldButton class

The definition of the **BaldButton** class begins in Listing 3 . This is a local class that extends **Button** .

Extending the **Button** class makes it possible to override the **processMouseEvent** method in order to handle mouse events that are fired by the button.

This is a form of low-level event handling that will be contrasted with source-listener event handling later in the program.

Listing 3 shows the constructor for the **BaldButton** class.

Enable mouse events

The most important code in the constructor is the statement that enables mouse events on the button.

If you are unfamiliar with the **enableEvents** method, you should look it up in the Sun documentation. Briefly, this method must be called on the button to cause the overridden **processMouseEvent** method to be called later when the button fires a mouse event.

The remaining constructor code

The remaining code in the constructor

- Sets the text value on the face of the button
- Gets and displays the name of the class file that represents this local class

The screen output

Construction of the button by the code in Listing 3 causes the text shown in Figure 5 to appear on the screen. This is how I was able to identify the name of the class file that represents the local class in my earlier discussion of class file names.

We will see later that this button will be added as the leftmost button in the GUI shown in Figure 2 .

The processMouseEvent method

Continuing with the constructor for the **GUI** class, Listing 4 shows the overridden **processMouseEvent** method for an object of the **BaldButton** class.

This method is called each time an object instantiated from this class fires a mouse event. That is why I refer to the method as an event handler for the button.

Different kinds of mouse events A button can fire a variety of different kinds or subcategories of **mouse events** :

- MOUSE_CLICKED
- MOUSE_DRAGGED
- MOUSE_ENTERED
- MOUSE_EXITED
- MOUSE_MOVED
- MOUSE_PRESSED
- MOUSE_RELEASED

In this case, I elected to ignore all but **MOUSE_CLICKED** . This subcategory of mouse event occurs when a mouse button is pressed and then released.

The code in the event handler of Listing 4 first confirms that the event was of the MOUSE_CLICKED variety. If so, it displays a message that matches the fifth line of text in the output shown in Figure 4 .

Call processMouseEvent on the superclass

Without getting into the details of why this is required, I'm simply going to tell you that when you use this low-level event model to handle events, your overridden **processMouseEvent** method must call the same method in the superclass, passing the incoming parameter of type **MouseEvent** as a parameter to the superclass version of the method.

Add a button to the frame

The last statement in Listing 4 instantiates a new **BaldButton** object, setting the text on the face of the button to **A** , and adds that new object to the frame.

Because the layout property of the frame has been set to FlowLayout , and because this is the first component added to the frame, this button appears as the leftmost button in the GUI shown in Figure 2 .

Could instantiate multiple buttons of this type

Although I instantiated the button object as an anonymous object in this case, that wasn't necessary. Using this local class, I could instantiate more than one object of this type, saving the object's references in reference variables of the appropriate type. Later we will see that this is not possible for anonymous classes.

It is interesting to note, however, that with this event handling model, if I were to instantiate multiple buttons of this type, the same **processMouseEvent** method would be called no matter which of the buttons fired a mouse event. If I wanted different behavior as a result of the different buttons firing mouse events, I would have to write code inside the **processMouseEvent** method to deal with that issue.

The source-listener event model that I will illustrate later doesn't suffer from that restriction.

An anonymous inner class for low-level event handling

Listing 5 shows the beginning of an anonymous class to perform low-level event handling similar to that shown in Listing 4 .

This code defines an anonymous inner class that implicitly extends **Button** and has mouse events enabled. I provided this class primarily for comparison with the local class named **BaldButton** . This class is an anonymous alternative to the local **BaldButton** class.

An argument to the add method

Note that the definition of this anonymous class appears as an argument to the **add** method for the frame. Thus, the anonymous object instantiated from the anonymous class is added as the second *(middle)* button in Figure 2 .

Extends the Button class

Note also that this form of anonymous class implicitly extends the **Button** class. Once again, this makes it possible to override the

processMouseEvent method belonging to the **Button** class.

An instance initializer

As I mentioned earlier in this module, it is not possible to define a constructor for an anonymous class. However, it is possible to define an instance initializer.

> *(In Listing 5 , the instance initializer consists of two statements enclosed by matching curly brackets.)*

This class defines an instance initializer that

- Enables mouse events on an anonymous object instantiated from the anonymous class.
- Gets and displays the name of the class file that represents the anonymous class.

The screen output

Therefore, the instantiation of this anonymous object causes the text shown in Figure 6 to appear on the screen. About all you can tell by looking at this class name is that it is the name of a file that represents an anonymous class.

Overridden processMouseEvent method

The remaining code in the anonymous class definition is shown in Listing 6 . The code in Listing 6 consists of

- an overridden **processMouseEvent** method, plus
- the curly brackets, parentheses, and semicolon necessary to complete the expression and the statement.

Same code as before

The code in this overridden **processMouseEvent** method is essentially the same as that shown for the local class in Listing 4 , except that it produces a different message on the screen when the user clicks the button.

Clicking the middle button in Figure 2 produces the screen output shown by the sixth line in Figure 4 .

Instantiating and registering a MouseListener object

Implementing a listener interface

Now I'm going to switch from ***low-level*** event handling to ***source-listener*** event handling. The code to accomplish this begins in Listing 7 .

With this event handling model:

- A listener object is instantiated from a class that implements a specific listener interface. In this case, that interface will be the **MouseListener** interface.
- The listener object is *registered* on an object that knows how to fire events of a type that is associated with the listener interface. In this case, that will be events of type **MouseEvent** .
- When the source object fires an event of the specified type, one of the methods belonging to the registered listener object will be called to handle the event. The different methods belonging to the listener object are declared in the implemented listener interface.

Listing 7 begins by instantiating a new **Button** object.

> *(Note that with this event model, it is not necessary to extend the **Button** class, because it is not necessary to override methods belonging to the **Button** object.)*

Register a MouseListener object

After instantiating the **Button** object, the code in Listing 7 calls the **addMouseListener** method to register a **MouseListener** object on that button. The argument to the **addMouseListener** method must be a reference to an object instantiated from a class that implements the **MouseListener** interface.

Instantiate the listener object

In this case, that listener object is created by

- writing an expression to instantiate an anonymous object from an anonymous class and
- placing that expression as an argument to the **addMouseListener** method.

Implement the MouseListener interface

The definition of the anonymous class in this example uses the syntax that implements an interface.

An instance initializer

As before, an instance initializer is used to get and display the name of the class file that represents the anonymous class. Thus, when the new anonymous object of the anonymous class is instantiated, the text shown in Figure 7 appears on the screen.

Note the similarity of this class file name to that shown earlier in Figure 6 . *(You can also compare them in Figure 4 .)* The names of the two class files differ only with respect to a number that is provided by the compiler to guarantee that each class file name is unique.

Implementing the interface

Whenever a class implements an interface, it must provide a concrete definition for each of the methods declared in the interface, even if some of those methods are empty.

Continuing with the definition of the anonymous class, Listing 8 provides definitions for all five of the methods declared in the **MouseListener** interface. Four of those methods are defined as empty methods.

Separation of event subcategories

One of the major differences between the *low-level* event model discussed earlier and the *source-listener* model being discussed here has to do with where the separation between the different subcategories *(mouseClicked, mousePressed, mouseReleased, etc.)* of a given event type is accomplished.

In the low-level model, the separation must be accomplished by code in the overridden event handler method, such as with the **if** statement in the **processMouseEvent** method defined in Listing 6 .

In the source-listener model, the separation is accomplished before the event handler method is called, and a specific event handler method, such as the **mouseClicked** method is called on the listener object.

When the button fires a mouse event ...

In this case, whenever the button fires a **MouseEvent** of the **MOUSE_CLICKED** subcategory , the **mouseClicked** method defined in Listing 8 will be called automatically, causing the seventh line of text in Figure 4 to appear on the screen.

Whenever the button fires a **MouseEvent** of one of the other subcategories , one of the empty methods defined in Listing 8 will be called. This method will return immediately, doing nothing but wasting a little computer time.

> *(In case you are wondering what happened to the **mouseMoved** and **mouseDragged** methods, they are defined in the **MouseMotionListener** interface instead of the **MouseListener** interface.)*

Add the button to the frame

Finally, the last statement in Listing 8 adds the new button to the **Frame** as the rightmost button in Figure 2 .

A disclaimer

I wrote this code the way that I did in Listing 8 to illustrate an anonymous class that implements an interface. In real life, I would probably cause the anonymous class to extend the **MouseAdapter** class and override the **mouseClicked** method instead of implementing the **MouseListener** interface. That would eliminate the requirement for me to define the four empty methods in Listing 8 .

Extending the WindowAdapter class

The above disclaimer provides a perfect lead-in to the definition of the anonymous class shown in Listing 9 .

Registering a WindowListener on the Frame

The code in Listing 9 instantiates an anonymous object of an anonymous class that extends the **WindowAdapter** class.

That anonymous object is registered as a **WindowListener** on the **Frame** by passing the object's reference to the **addWindowListener** method belonging to the **Frame** .

> *(The **addWindowListener** method requires an incoming parameter of type **WindowListener** . This is satisfied by the fact that the **WindowAdapter** class implements the **WindowListener** interface. Thus, an object instantiated from a class that extends **WindowAdapter** can also be treated as type **WindowListener** .)*

The screen output

This anonymous class definition uses an instance initializer to get and display the name of the class that represents the anonymous class. Thus, when the anonymous object of the anonymous class is instantiated, the text shown in Figure 8 appears on the screen.

> *(In an earlier module explaining member classes, I told you that it is possible to examine the names of the class files that represent the member classes and to determine quite a lot about the structure of the program in terms of which classes are members of which other classes. However, in the case of local classes and anonymous classes, about all that you can determine from the name of the class file is that the file either represents a local class or represents an anonymous class (see the summary of class names in Figure 3).)*

The windowClosing method

The code in Listing 9 overrides the **windowClosing** method inherited from the **WindowAdapter** class.

Clicking the *close* button with the X in the upper right hand corner of Figure 2 causes the **windowClosing** method to be called on any **WindowListener** objects that have been registered on the frame.

In this case, the overridden **windowClosing** method in Listing 9 causes the last line of text in Figure 4 to be displayed on the screen.

Following that, the overridden **windowClosing** method calls the **System.exit** method to terminate the program.

The remaining code

The remaining code in Listing 9

- Causes the frame to become visible
- Signals the end of the constructor

- Signals the end of the **GUI** class

The GUI remains on the screen until terminated

Once the constructor is executed, the GUI simply remains on the screen waiting for someone to click one of the buttons or to click the close button in the upper right corner of the frame. When these buttons are clicked, the event handlers are called, causing text such as that shown in Figure 9 to appear on the screen.

Simple event handlers

In this demo program, the event handlers simply display messages on the screen, and in the case of the close button, terminate the program. In a real world program, the behavior of the event handlers would likely be much more substantive, but the overall skeleton of the program need not be any different from that illustrated here.

Run the program

I encourage you to copy the code from Listing 10 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In addition to a number of other items, a class definition can contain:

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

Member classes and local classes were explained in previous modules. This module explains anonymous classes.

Although there are some differences, an anonymous class is very similar to a local class without a name. Instead of defining a local class and then instantiating it, you can often use an anonymous class to combine these two steps.

An anonymous class is defined by a Java expression, not a statement. Therefore, an anonymous class definition can be included within a larger overall Java expression.

Anonymous class definitions are often included as arguments to method calls, or as the right operand of assignment operators.

An object of an anonymous class must be internally linked to an object of the enclosing class.

There is no such thing as an anonymous interface, a local interface, or a member interface.

An anonymous class can often be defined very close to its point of use. Once you become accustomed to the somewhat cryptic syntax used with anonymous classes, this can lead to improved code readability.

Probably the most important benefit of anonymous classes has to do with accessing the members of enclosing classes. As with member classes and local classes, methods of an anonymous class have direct access to all the members of the enclosing classes, including private members. Thus the use of anonymous classes can sometimes eliminate the requirement to connect objects together via constructor parameters. In addition, objects of anonymous classes have access to **final** local variables that are declared within the scope of the anonymous class.

An anonymous class can be particularly useful in those cases where

- There is no reason for an object of the anonymous class to exist in the absence of an object of the enclosing class.

- There is no reason for an object of the anonymous class to exist outside a method of the enclosing class.
- Methods of the object of the anonymous class need access to members of the object of the enclosing class.
- Methods of the object of the anonymous class need access to **final** local variables and method parameters belonging to the method in which the anonymous class is defined.
- Only one instance of the anonymous class is needed.
- There is no need for the class to have a name that is accessible elsewhere in the program.

An anonymous class cannot define a constructor. However, it can define an instance initializer.

Any arguments that you specify between the parentheses following the superclass name in an anonymous class definition are implicitly passed to the superclass constructor.

Only one instance of an anonymous class can be created.

As with member classes and local classes, anonymous classes cannot contain **static** members.

As with local variables and local classes, anonymous classes cannot be declared **public** , **protected** , **private** , or **static** .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Anonymous Classes
- File: Java1640.htm
- Published: 11/20/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is show in Listing 10 .

Figures

Figure 1 - Syntax for anonymous classes.

Figure 1 - Syntax for anonymous classes.

```
new className(optional argument list)
{classBody}

new interfaceName(){classBody}
```

Note:

Figure 2 - Program GUI.

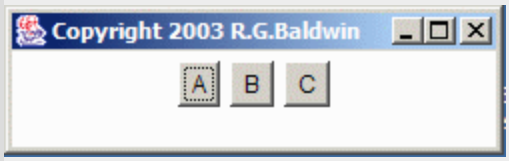


Figure 3 - Class file names.

```
GUI$1$BaldButton.class
GUI$1.class
GUI$2.class
GUI$3.class
GUI.class
InnerClasses08.class
```

Figure 4 - Screen output.

```
Local class name: GUI$1$BaldButton
Anonymous class B name: GUI$1
Anonymous class C name: GUI$2
Anonymous window listener class name: GUI$3
buttonA clicked
buttonB clicked
buttonC clicked
Close button clicked
```

Figure 5 - Screen output.

```
Local class name: GUI$1$BaldButton
```

Figure 6 - Screen output.

```
Anonymous class B name: GUI$1
```

Figure 7 - Screen output.

```
Anonymous class C name: GUI$2
```

Figure 8 - Screen output.

```
Anonymous window listener class name: GUI$3
```

Figure 9 - Screen output.

```
buttonA clicked
buttonB clicked
buttonC clicked
Close button clicked
```

Listings

Listing 1 - The controlling class.

```
public class InnerClasses08 {
  public static void main(String[] args){
    new GUI();
  }//end main
}//end class InnerClasses08
```

Listing 2 - Beginning of the GUI class.

```
class GUI extends Frame{
  public GUI(){//constructor
    setLayout(new FlowLayout());
    setSize(250,75);
    setTitle("Copyright 2003 R.G.Baldwin");
```

Listing 3 - Beginning of the BaldButton class.

Listing 3 - Beginning of the BaldButton class.

```
    class BaldButton extends Button{
      BaldButton(String text){//constructor

enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        setLabel(text);
        System.out.println("Local class name:
" +

getClass().getName());
      }//end constructor
```

Listing 4 - The processMouseEvent method.

Listing 4 - The processMouseEvent method.

```
      public void processMouseEvent(MouseEvent
e){
        if (e.getID() ==
MouseEvent.MOUSE_CLICKED){
          System.out.println("buttonA
clicked");
        }//end if

        //The following is required of
overridden
        // processMouseEvent method.
        super.processMouseEvent(e);
      }//end processMouseEvent
    }//end class BaldButton

    //Add button to Frame
    add(new BaldButton("A"));
```

Listing 5 - Beginning of anonymous inner class.

Listing 5 - Beginning of anonymous inner class.

```
    add(new Button("B")
      {//Begin class definition
        {//Instance initializer
          enableEvents(

AWTEvent.MOUSE_EVENT_MASK);
          System.out.println(
                      "Anonymous class B name:
" +

getClass().getName());
        }//end instance initializer
```

Listing 6 - Overridden processMouseEvent method.

Listing 6 - Overridden processMouseEvent method.

```
        public void processMouseEvent(
                                        MouseEvent
e){
          if (e.getID() ==

MouseEvent.MOUSE_CLICKED){
            System.out.println("buttonB
clicked");
          }//end if

          //Required of overridden
          // processMouseEvent method.
          super.processMouseEvent(e);
        }//end processMouseEvent
      }//end class definition
    );//end add method call
```

Listing 7 - Register a MouseListener object.

Listing 7 - Register a MouseListener object.

```
    Button buttonC = new Button("C");

    buttonC.addMouseListener(new
MouseListener()
      {//begin class definition
        //Instance initializer
        {System.out.println(
                     "Anonymous class C name:
" +

getClass().getName());}
```

Listing 8 - Implementing the interface.

Listing 8 - Implementing the interface.

```
        public void mouseClicked(MouseEvent e)
{
          System.out.println("buttonC
clicked");
        }//end mouseClicked

        //All interface methods must be
defined
        public void mousePressed(MouseEvent e)
{}
        public void mouseReleased(MouseEvent
e){}
        public void mouseEntered(MouseEvent e)
{}
        public void mouseExited(MouseEvent e)
{}

      }//end class definition
    );//end addMouseListener call

    add(buttonC);//add button to frame
```

Listing 9 - Registering a WindowListener on the Frame .

Listing 9 - Registering a WindowListener on the Frame .

```
    addWindowListener(new WindowAdapter()
      {//begin class definition
        //Instance initializer
        {System.out.println(
             "Anonymous window listener class
" +
               "name: " +
getClass().getName());}

        public void windowClosing(WindowEvent
e){
          System.out.println(
                          "Close button
clicked");
          System.exit(0);
        }//end windowClosing
      }//end class definition
    );//end addWindowListener

    setVisible(true);

  }//end constructor
}//end GUI class
```

Note:

Listing 10 - Complete program listing.

```
/*File InnerClasses08.java
Copyright 2003 R.G.Baldwin

This program is designed to illustrate the use
```

```
of local classes, and anonymous classes.  It
illustrates three different implementations of
anonymous classes.  It also illustrates the use
of instance initializers as an alternative to
constructors.

Illustrates use of local class to instantiate
object to handle mouse clicked event with
low-level event handling.  This class uses
constructor to enable mouse events on a new
extended Button class.  Also uses constructor
to display the class file name.

Illustrates use of anonymous class to instantiate
object to handle mouse clicked event with
low-level event handling.  This class uses an
instance initializer to enable mouse events on a
new extended Button class.  Also uses instance
initializer to display name of class file.

Illustrates use of anonymous class, which
implements MouseListener interface, to
instantiate object to handle mouse clicked event
using source-listener event model.  Uses instance
initializer to display name of class file.

Illustrates use of anonymous class, which extends
WindowAdapter class, to instantiate object to
handle window events fired by the close button in
the upper-right corner of a Frame object, using
source-listener event model.  Uses instance
initializer to display name of class file.

This program produces the following class files
when compiled:

GUI$1$BaldButton.class
```

```
GUI$1.class
GUI$2.class
GUI$3.class
GUI.class
InnerClasses08.class

The program produces the following output when
the program is started, each button is clicked
once in succession, and then the close button
in the upper-right corner of the Frame is
clicked:

Local class name: GUI$1$BaldButton
Anonymous class B name: GUI$1
Anonymous class C name: GUI$2
Anonymous window listener class name: GUI$3
buttonA clicked
buttonB clicked
buttonC clicked
Close button clicked

Tested using JDK 1.4.1 under Win
**********************************************/

import java.awt.*;
import java.awt.event.*;

public class InnerClasses08 {
  public static void main(String[] args){
    new GUI();
  }//end main
}//end class InnerClasses08
//===========================================//

class GUI extends Frame{

  public GUI(){//constructor
```

```
    setLayout(new FlowLayout());
    setSize(250,75);
    setTitle("Copyright 2003 R.G.Baldwin");

    //Local class w/mouse events enabled. The new
    // class extends Button, and uses low-level
    // event handling to handle mouse clicked
    // events on the button.
    class BaldButton extends Button{
      BaldButton(String text){//constructor
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        setLabel(text);
        //Display the name of the class file
        System.out.println("Local class name: " +
                           getClass().getName());
      }//end constructor

      //This is the event handling method.
      public void processMouseEvent(
                                    MouseEvent e){
        if (e.getID() ==
                        MouseEvent.MOUSE_CLICKED){
          System.out.println("buttonA clicked");
        }//end if
        //The following is required of overridden
        // processMouseEvent method.
        super.processMouseEvent(e);
      }//end processMouseEvent
    }//end class BaldButton

    //Add button to Frame
    add(new BaldButton("A"));


    //This code defines an anonymous Inner Class
    // w/mouse events enabled.  The new class
    // extends Button.  This class uses low-level
```

```
    // event handling to handle mouse clicked
    // events on the button.  This is an
    // anonymous alternative to the local class
    // defined above.
    add(new Button("B")
      {//Begin class definition
        {//Instance initializer
          enableEvents(
                      AWTEvent.MOUSE_EVENT_MASK);
          System.out.println(
                     "Anonymous class B name: " +
                          getClass().getName());
        }//end instance initializer

        //Override the inherited
        // processMouseEvent method.
        public void processMouseEvent(
                                    MouseEvent e){
          if (e.getID() ==
                       MouseEvent.MOUSE_CLICKED){
            System.out.println(
                                "buttonB clicked");
          }//end if
          //Required of overridden
          // processMouseEvent method.
          super.processMouseEvent(e);
        }//end processMouseEvent
      }//end class definition
    );//end add method call


    Button buttonC = new Button("C");
    //Anonymous inner class that implements
    // MouseListener interface
    buttonC.addMouseListener(new MouseListener()
      {//begin class definition
        //Instance initializer
```

```
      {System.out.println(
                  "Anonymous class C name: " +
                       getClass().getName());}

      public void mouseClicked(MouseEvent e){
        System.out.println("buttonC clicked");
      }//end mouseClicked

      //All interface methods must be defined
      public void mousePressed(MouseEvent e){}
      public void mouseReleased(MouseEvent e){}
      public void mouseEntered(MouseEvent e){}
      public void mouseExited(MouseEvent e){}

    }//end class definition
  );//end addMouseListener call

  add(buttonC);//add button to frame


  //Use an anonymous class to register a window
  // listener on the Frame.  This class extends
  // WindowAdapter
  addWindowListener(new WindowAdapter()
    {//begin class definition
      //Instance initializer
      {System.out.println(
           "Anonymous window listener class " +
             "name: " + getClass().getName());}

      public void windowClosing(WindowEvent e){
        System.out.println(
                        "Close button clicked");
        System.exit(0);
      }//end windowClosing
    }//end class definition
  );//end addWindowListener
```

```
    setVisible(true);

  }//end constructor

}//end GUI class
//=============================================//
```

-end-

Java1642 The Essence of OOP using Java, Nested Top-Level Classes
Baldwin explains nested top-level classes, and illustrates a very useful polymorphic structure where nested classes extend the enclosing class and override methods declared in the enclosing class.

Revised: Wed May 11 16:20:00 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled ITSE2317 - Java Programming (Intermediate) that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1642 The Essence of OOP using Java, Nested Top-Level Classes
- File: Java1642.cnx.htm
- Published: 01/13/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4210: Getting Started with Generics
This module presents and explains several simple programs that provide a first look into Java Generics.

Revised: Wed May 11 17:07:35 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- General background information
- Preview
- Discussion and sample code
    - Collection behavior prior to Java version 1.5
    - An incorrect cast in code that doesn't use generics
    - Dealing with angle brackets in cnxml body text
    - Avoiding the requirement to cast through the use of generics
    - Compile-time type safety provided by the use of generics
    - Generic iterator syntax and the enhanced for loop
- Run the programs
- Summary
- What's next?
- Miscellaneous
- Complete program listings

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

Many new features in Java version 1.5

When Java version 1.5 was released, it contained many new language features, including:

- Generics
- Enhanced for Loop
- Autoboxing/Unboxing
- Typesafe Enums
- Varargs
- Static Import
- Metadata

In addition to the new language features, Java version 1.5 also contained other new features such as new *Look and Feel* capabilities for Swing GUIs.

First in a series

This is the first module in a series of modules designed to teach you about Generics in Java version 1.5. This module will teach you some of the rudimentary aspects of the generics capability . This module will also teach you how to use the *enhanced **for** loop* with collections.

Future modules will teach you how to use other aspects of generics

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

Figures

- Figure 1 . Typical v1.5 compiler warning.
- Figure 2 . Warning with -Xlint:unchecked.
- Figure 3 . A ClassCastException.
- Figure 4 . A compiler error.
- Figure 5 . Iterator output.

Listings

- Listing 1 . Beginning of the program named Generics05.
- Listing 2 . Beginning of the runIt method.
- Listing 3 . An iterator.
- Listing 4 . Enhanced for loop.
- Listing 5 . The program named Generics01.
- Listing 6 . The program named Generics02.
- Listing 7 . The program named Generics03.
- Listing 8 . The program named Generics04.
- Listing 9 . The program named Generics05.

General background information

A new compiler warning

Have you recently tried to recompile a legacy program that has compiled and executed properly for many years only to get a compiler message similar to that shown in Figure 1 ?

Figure 1 - Typical v1.5 compiler warning.

Figure 1 - Typical v1.5 compiler warning.

```
Note: Generics01.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for
details.
```

If you followed up and recompiled as directed, you probably got a message from the compiler that looked something like the one shown in Figure 2 .

Figure 2 - Warning with -Xlint:unchecked.

```
Generics01.java:34: warning: [unchecked]
unchecked call to add(E) as a member of
 the raw type ArrayList
    var1.add(new Date());
             ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
1 warning
```

If that has happened to you, you have had your first encounter with a new feature in Java version 1.5 known as *generics.*

What are generics?

Oracle has this to say about generics *(boldface added for emphasis):*

Note:
"This long-awaited enhancement to the type system allows a type or method to operate on objects of various types while providing ***compile-time type safety*** *. It adds compile-time type safety to the Collections Framework and eliminates the drudgery of casting."*

In addition to the Collections Framework, generics impacts several other areas of Java programming as well.

Generics are not particularly easy

Don't be lulled into a false sense of security by what you will find in this module. Although the introductory material presented in this module is rather straightforward, a full understanding of generics can be fairly difficult. The tentacles of generics reach into many different areas of Java in very complex ways.

The truth of this is borne out by the fact that the excellent book entitled Java How to Program, Sixth Edition (Deitel) dedicates an entire chapter consisting of more than 30 pages to generics. Furthermore, the chapter on generics doesn't even include many additional pages that the book dedicates to a discussion of the impact of generics on the Java Collections Framework. The book covers several topics in the chapter on generics, including the following:

- Generic methods
- Generic classes
- Raw types
- Wildcards
- Generics and inheritance

The impact of generics on the Java Collections Framework is covered in a different chapter. Hopefully, the benefits provided by generics will outweigh the additional complexity. Only time will tell.

The Java Collections Framework

Even though the material that will be presented in this module will be relatively straightforward, you will need to know something about the Java Collections Framework for it to make much sense. If you need to learn more about the Java Collections Framework, see the modules beginning with Java4010: Getting Started with Java Collections at I TSE 2321 - Object-Oriented Programming (Java) . Be aware, however, that those modules were originally published long before generics existed in Java. If you compile the programs in those modules using Java version 1.5 or a later version, you will get the warning messages from the compiler shown in Figure 1 .

Preview

I will present and explain several very simple programs in this module. Those programs are intended to illustrate the following concepts:

- How the Java Collections Framework behaved prior to the release of Java version 1.5.
- The effect of an incorrect cast in code that doesn't use generics.
- Avoiding the requirement to cast through the use of generics. This program also includes an illustration of some of the required syntax for generics.
- Compile-time type safety provided by the use of generics.
- Syntax requirements for the use of iterators with generics. This program also illustrates the use of the enhanced **for** loop with collections, which is another new feature in Java version 1.5.

Discussion and sample code

Listings of the programs

Complete listings of all the programs discussed in this module are provided in Listing 5 through Listing 9 near the end of the module.

Collection behavior prior to Java version 1.5

The main purpose of the program named **Generics01** is to establish a baseline against which to compare the other programs. A secondary purpose is to illustrate the warnings produced by the Java version 1.5 *(or later)* compiler when the syntax of the source code doesn't take generics into account.

A complete listing of the program named **Generics01** is shown in Listing 5 near the end of the module.

As you will see later, this program was written using the program syntax and style that was common prior to the release of Java version 1.5. In particular, this program does not include the syntax necessary to take generics into account.

Notes at compile time

When this program is compiled using the Java version 1.5 or later compiler, the text shown in Figure 1 appears on the screen.

Note that the text in Figure 1 is not identified as either an error or a warning. Rather, the text is identified simply as notes. These notes provide instructions on how to recompile and get more information regarding a potential problem.

Recompiling with the Xlint switch

When the program is recompiled using the Xlint switch shown in Figure 1 , the compiler produces the text shown in Figure 2 . Note that this text is identified as a warning.

Note:
*(Later when we examine the code from the program named **Generics01** , you can compare it with this text to see just what the compiler is complaining about.)*

Get used to it

The text in Figure 1 and Figure 2 is similar to what you can expect to see any time that you use the Java version 1.5 or later compiler to compile a program that includes the Java Collections Framework and doesn't use the required syntax to take generics into account. This probably includes many of the programs that you wrote, compiled, and executed successfully prior to the release of Java version 1.5.

The program code

The code for the program named **Generics01** is shown in its entirety in Listing 5 .

The main method

As you can see in Listing 5 , the **main** method instantiates a new object of the class named **Generics01** , and calls the method named **runIt** on that object. It is the behavior of the **runIt** method that interests us. More particularly, it is the expression contained in the argument list for the **println** method that interests us the most.

A baseline program

As mentioned earlier, the main purpose of this program is to establish a baseline against which we can compare the other programs to be discussed later. This program was written exactly as it would have been written prior to the release of generics in Java version 1.5.

An ArrayList object

The program declares and initializes an instance variable named **var1** with a reference to an object instantiated from the class named **ArrayList** . **ArrayList** is one of the concrete implementations of the interfaces provided in the Java Collections Framework. Briefly, an object of the **ArrayList** class is an object that implements the **List** interface, providing a convenient place to store references to other objects.

One of the methods of an **ArrayList** object is the method named **add** . This method is used to add new elements to the end of the list.

Adding a Date object's reference to the list

Listing 5 calls the **add** method to add a new **Date** object's reference to the list.

Note:
(This is the method call that the compiler was complaining about in the warning in Figure 2 .)

When an object's reference is added to an **ArrayList** object *(without the use of generic syntax)*, that reference is automatically converted to and stored as type **Object** .

Note:
(As you will see later, that is probably also true even with the use of generic syntax. I will have more to say about this later.)

What can you do with a reference of type Object?

I often tell my students that there are only twelve things that you can do with an object's reference that has been converted to type **Object** . The first eleven of those twelve things is to call any one of the eleven methods that are defined in the **Object** class and inherited into all subclasses of **Object** . The twelfth thing is to attempt to cast the reference to some other type in an attempt to call some other method on the reference after casting.

Invoking getTime on the reference

The objective of the code in this program is to call the method named **getTime** on the **Date** object's reference. The **getTime** method is not one of the eleven methods that are defined in the **Object** class. Rather, the **getTime** method is defined in the **Date** class.

Change the type of the reference

Therefore, in order to call the **getTime** method on the reference after it is retrieved from the **ArrayList** object, the type of the reference must be converted from type **Object** to type **Date** . This is accomplished by the cast operation in the expression contained in the argument list for the **println** method in Listing 5 .

That expression calls the **get** method on the reference to the **ArrayList** object to retrieve the element stored at index 0 of the collection. The **get** method returns the reference as type **Object** *(at least that was true prior to the introduction of generics).* Then the expression casts the reference to type **Date** , converting its type back to type **Date** .

Invoking the getTime method

Finally, the expression successfully calls the **getTime** method on the reference of type **Date** .

The program produces an output similar to that shown below:

1377991758665

This is the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by the **Date** object.

Casting was a necessity prior to Java version 1.5

Prior to the release of Java version 1.5, it was always necessary to cast references retrieved from collection objects in order to call any methods on them other than the eleven methods defined in the **Object** class.

Some authors refer to this casting requirement as *"the drudgery of casting,"* and indicate that casting may be eliminated through generics.

Note:
(In my opinion, from this viewpoint alone, the cure is worse than the disease. Casting syntax is much simpler and more straightforward than generics

syntax.)

May eliminate runtime errors and exceptions

However, it is possible for the programmer to perform an incorrect cast at this point in the program, which will usually result in a **ClassCastException** being thrown at runtime.

The great promise of generics is that it can sometimes cause programming errors to be recognized at compile time instead of encountering them at runtime.

Obviously, the best approach is to avoid writing programs containing programming errors in the first place. However, if you are going to write programs containing errors, it is usually better to catch them at compile time than to have them occur at runtime. This is the thing that may make the complexity of generics worthwhile.

An incorrect cast in code that doesn't use generics

The program named **Generics02** *(shown later in Listing 6)* illustrates the application of an incorrect cast to an element that is retrieved from an **ArrayList** object, along with the runtime error that is produced by that incorrect cast.

No compiler error

This program does not produce a compiler error.

Note:
(However, the Java version 1.5 compiler does produce a warning having to do with the failure to apply the new generics syntax that was released in Java version 1.5. Note that earlier compilers would not have produced such a warning.)

The point here is that the compiler does not check to confirm that the correct cast is applied. This results in a successful compilation, but the program throws an exception at runtime.

A ClassCastException

The runtime exception that is thrown by this program is shown in Figure 3 . You can compare the details of this exception with the program code later. The reference to line 46 in Figure 3 is a reference to the statement in Listing 6 that attempts to cast the reference to type **Date** . *(That reference was originally a reference to an object of type **String** .)*

Figure 3 - A ClassCastException.

```
Exception in thread "main"
java.lang.ClassCastException:
 java.lang.String cannot
 be cast to java.util.Date
        at Generics02.runIt(Generics02.java:46)
        at Generics02.main(Generics02.java:53)
```

The program code

The program is shown in its entirety in Listing 6 . As before, the **main** method instantiates an object of the **Generics02** class and calls the **runIt** method on that object. Also as in the previous program, this program instantiates a new **ArrayList** object and saves the object's reference in the instance variable named **var1** .

The runIt method

The **runIt** method begins by populating the **ArrayList** object with references to two different objects of different types. One of the objects is type **Date** . The other object is type **String** .

A questionable programming style

While this is probably not a very good programming style, it is a style that was commonly used by Java programmers prior to the advent of generics. It was common to populate collection objects with references to a mixture of objects of different types.

References are stored as type Object

As you already know, when these object's references are put into the collection, the types of all the references are automatically converted to type **Object** . As you also already know, when the references are retrieved from the collection, if the purpose is to call any method on a reference other than one of the eleven methods defined in the **Object** class, it is necessary to cast the reference to a type that is consistent with the method.

Casting errors are likely

This is a scenario where the programmer is likely to make a casting error when casting elements retrieved from the collection. Unless the programmer uses the **instanceof** operator to determine the type of a retrieved reference prior to performing the cast, the programmer is depending on his memory to know the type of each reference in the population on the basis of the index of the element. If the programmer loses track of the types of the different references with respect to the element's indices, a casting error is a strong possibility.

The error scenario

This is the error scenario depicted by the first call to the **println** method in Listing 6 . In this case, the element at index 0 is a reference to a **Date** object, and the element at index 1 is a reference to a **String** object. However, the programmer mistakenly retrieves the element at index 1 and attempts to cast it to type **Date** , which is the type of the reference at index 0. This results in the runtime exception shown in Figure 3 .

Of course, it has always been possible to use the **instanceof** operator to confirm the type of a reference before performing a cast as a way to avoid this type of programming error. Good programming practice would dictate the use of that construct when working with references to objects of mixed types in a single collection.

Dealing with angle brackets in cnxml body text

You are reading this module at cnx.org. Documents published at cnx.org are maintained in a special flavor of xml know as cnxml. Without getting into the details, I will tell you that creating cnxml is not an easy task. In order to avoid having to write these modules in raw cnxml code, I originally write them using a WYSIWYG xhtml editor and then run the xhtml files through a program of my own design that transforms the xhtml document into cnxml.

Unfortunately, my transform program is incapable of handling angle brackets in the body text of the document. Therefore, whenever the body text in this document needs a left angle bracket, I will indicate the existence of a left or right angle bracket using a text description of the angle bracket. The code in the listings, however, will show the left and right angle brackets intact as required.

Avoiding the requirement to cast through the use of generics

The program named **Generics03** shown in Listing 7 illustrates how generics can be used to avoid the requirement to cast references when they are retrieved from a collection.

Note:
(While casting is not difficult, avoiding the requirement to cast can also avoid the possibility of casting incorrectly.)

As in the previous programs, the **main** method in this program instantiates an object of the class named **Generics03** and calls the method named **runIt** on the object.

Instantiate an ArrayList object

Also, as in the previous programs, this program declares an instance variable named **var1** and initializes that variable with a reference to a new object of type **ArrayList** . However, the syntax that is used for this purpose in this program is significantly different from the syntax used for the same purpose in the previous two programs.

Note:
*(Note the code in Listing 7 that shows the type **Date** enclosed in matching angle brackets. This is the primary syntax change required to use generics.)*

What does this syntax mean?

One way to think of this syntax is that the expression on the right of the assignment operator instantiates a new **ArrayList** object that is capable of containing only references to objects of type **Date** .

Note:
*(Those references are probably still stored as type **Object** . We will see how this apparent discrepancy is reconciled later through automatic casting.)*

Similarly, the expression on the left of the assignment operator is the declaration of an instance variable capable of holding a reference to an **ArrayList** object, which in turn is capable of containing only references to objects of type **Date** .

Must qualify both expressions

Additional changes to generics were made in the release of Java version 1.7. Prior to the release of Java version 1.7, it was necessary to qualify the expressions on both sides of the assignment operator by use of the angle-bracket syntax. If the qualifier was included in the expression on the right, but was omitted from the variable declaration on the left, the compilation would fail later. That is still true following the release of Java version 1.7.

If the angle-bracket qualifier was included with the variable declaration on the left and omitted from the instantiation of the new object on the right, the program would compile and run successfully. However, the compiler would issue an *unchecked conversion* warning indicating the possibility of a runtime error under certain conditions.

Following the release of Java version 1.7, it is not necessary to qualify the expression on the right side of the assignment operator. You will learn more about this in a future module.

What does Oracle have to say?

According to Oracle *(boldface added for emphasis)*,

> **Note:**
> *"Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and* ***can insert the correct casts*** *on values being taken out of the collection."*

Note the boldface text in the above quotation indicating that the compiler modifies your code by inserting casts where appropriate.

Compile-time type safety

In discussing code similar to the code in Listing 7 , Oracle states,

Note:
"... so the compiler can verify at compile time that the type constraints are not violated at run time. Because the program compiles without warnings, we can state with certainty that it will not throw a ***ClassCastException*** *at run time. The net effect of using generics, especially in large programs, is improved readability and robustness."*

In order to achieve compile-time type safety, it is necessary that the program compiles without warnings. Otherwise, the program may execute, but may throw a **ClassCastException** at runtime.

More on generics with collections

In further explaining generics as used with collections, Oracle goes on to say *(boldface added for emphasis):*

Note:
"... when we declare c to be of type Collection (String in angle brackets), this tells us something about the variable c that holds true wherever and whenever it is used, and the compiler guarantees it (assuming the program ***compiles without warnings*** *). A cast, on the other hand, tells us something the programmer thinks is true at a single point in the code, and the VM checks whether the programmer is right only at run time."*

The bottom line

The bottom line on generics *(when used with collections)* seems to be that references to objects are still stored in the collection as type **Object** . However, when we notify the compiler of the type of data to be stored in the collection using angle-bracket syntax, and the program compiles without warnings, the compiler will do at least the following:

- Ensure that only references to objects of the specified type are stored in the collection, and used consistently throughout the program, thus eliminating the possibility of a **ClassCastException** at runtime.
- Automatically cast the reference to the specified type when it is later retrieved by program code.

No explicit cast is required

That brings us back to a discussion of the code in Listing 7 . Note that unlike the code in Listing 5 , the print statement in Listing 7 does not contain an explicit cast to type **Date** , *(at least not in the code that I wrote).*

As described above, having been notified that the collection can contain only references to objects of type **Date** , the compiler automatically inserted a cast to type **Date** at the appropriate place in the code, thereby guaranteeing that the reference is converted from type **Object** to type **Date** before the **getTime** method is called on the reference.

There is still a cast involved. However, the cast is automatically inserted by the compiler. This eliminates the requirement for me *(the programmer*) to insert the cast, and also eliminates the possibility of me inserting an incorrect cast.

Once again, all of this assumes that the program compiles without warnings.

Program output

The program in Listing 7 compiles and executes correctly, producing the following output for one particular run.

1377995720768

Compile-time type safety provided by the use of generics

The program named **Generics04** shown in Listing 8 illustrates the ability of generics to prevent the storing of the wrong type of references in a collection. This in turn can prevent runtime errors.

A new ArrayList object

Listing 8 instantiates a new object of type **ArrayList** capable of storing only references to objects of type **Date** . This object's reference is stored in the instance variable named **var1** .

Other types cannot be stored in the collection

Once the **ArrayList** has been constrained to contain only references to objects of type **Date** , the compiler will not allow a reference to an object of any other type *(other than types that are assignment compatible with **Date** , such as subclasses of **Date**)* to be stored in the collection.

A compiler error rather than a runtime error

The first statement in the **runIt** method in Listing 8 attempts to add a new element to the **ArrayList** object. The new element is a reference to a literal **String** object that encapsulates the string "abcd". This results in the compiler error shown in Figure 4 .

Figure 4 - A compiler error.

Figure 4 - A compiler error.

```
Generics04.java:34: error: no suitable method
found for add(String)
    var1.add("abcd");
        ^
    method ArrayList.add(int,Date) is not
applicable
      (actual and formal argument lists differ
in length)
    method ArrayList.add(Date) is not applicable
      (actual argument String cannot be
converted to Date by method invocation
      conversion)
1 error
```

Without the use of generics, a reference to an object of any type could be added to the collection. This could result in a runtime error later if the programmer expected an object of type **Date** when in fact the object is of type **String** . If you are going to write programs containing errors, compiler errors are almost always preferable to runtime errors.

Generic iterator syntax and the enhanced for loop

Iterators

Listing 7 and Listing 8 showed the syntax that you must use to cause a collection object to be treated as a generic collection. You must also use a special syntax when working with generic iterators and the Java Collections Framework.

The enhanced for loop

Another new feature of Java version 1.5, referred to by Oracle as an *enhanced **for** loop*, can be used in certain situations to provide most of the benefits of an iterator with a somewhat simpler syntax.

Note:
(The enhanced for loop is also sometimes referred to as a for-each loop.)

The program named **Generics05** , which begins in Listing 1 , illustrates both of these concepts. A complete listing of the program is provided in Listing 9 .

Listing 1 - Beginning of the program named Generics05.

```
import java.util.*;

public class Generics05{
  //Create an ArrayList object suitable for
  // storing references to Date objects.
  ArrayList <Date> var1 = new ArrayList<Date>();
```

The main method

As in the previous programs, the **main** method instantiates an object of the **Generics05** class and calls the **runIt** method on that object. You can view the **main** method in Listing 9 near the end of the module.

Instantiate a generic ArrayList object

This program is a little longer than the previous programs, so I will break it down and explain it in fragments.

Listing 1 shows the beginning of the **Generics05** class. The code in Listing 1 instantiates a new **ArrayList** object capable of storing references to objects of type **Date** only. The code in Listing 1 also saves that object's reference in a generic instance variable named **var1** . This is the same syntax that you have seen in previous listings.

Populate the collection

Listing 2 shows the beginning of the **runIt** method. This code populates the **ArrayList** object with references to three **Date** objects. The first object encapsulates the current date and time. The second object encapsulates the date and time one day later than the first. The third object encapsulates the date and time two days later than the first object.

Listing 2 - Beginning of the runIt method.

```
void runIt(){
  //Get current date and time in milliseconds.
  long now = new Date().getTime();
  //Get length of one day in milliseconds
  long oneDay = 24 * 60 * 60 * 1000;

  //Populate the ArrayList object
  var1.add(new Date(now));
  var1.add(new Date(now + oneDay));
  var1.add(new Date(now + 2 * oneDay));
```

The code in Listing 2 is straightforward and shouldn't require further explanation.

Get and use an iterator

An iterator is an object instantiated from a specially designed class that implements the **Iterator** interface. The design of the class makes it possible for client code to gain sequential access to each element encapsulated in an associated collection object without a requirement to know anything about how the collection is structured.

Required syntax

The first statement in Listing 3 shows the syntax required to get and save a reference to a generic iterator for the **ArrayList** object instantiated earlier in Listing 1 .

Listing 3 - An iterator.

```
//Get an iterator
Iterator <Date> iter = var1.iterator();

//Perform the iteration
while(iter.hasNext()){
  System.out.println(iter.next().getTime());

}//end while loop

System.out.println();//blank line
```

Note the requirement to qualify the declaration of the local variable named **iter** with the type of data stored in the collection using angle-bracket notation similar to that used earlier.. You might think of this as a variable capable of holding a reference to an iterator object, which is capable of iterating on an **ArrayList** object, which in turn is capable of storing references to objects of type **Date** only.

Perform the iteration

The remaining code in Listing 3 uses the iterator to sequentially access and display information encapsulated in each of the three **Date** objects whose references are stored in the **ArrayList** object. This is standard code for the use of an iterator and should not require further explanation. This code produces the first three lines of text *(plus the blank line)* shown in Figure 5 .

Figure 5 - Iterator output.

```
1378070280877
1378156680877
1378243080877

1378070280877
1378156680877
1378243080877
```

The program output

The output produced by this program depends on when you run it. The output produced for one particular run is shown in Figure 5 .

The output will be different each time you run the program depending on the current date and time.

The enhanced for loop

The code in Listing 4 performs the same iteration using the new enhanced **for** loop that was released in Java version 1.5.

Listing 4 - Enhanced for loop.

```
    //Now perform the same iteration using
    // the new for-each construct.
    for(Date element : var1){
      System.out.println(element.getTime());
    }//end for-each

  }//end runIt
```

You might think of this syntax as meaning:

Note:
*For each element of type **Date** contained in the collection referred to by **var1** , get the value of the element and save it in the variable named **element** . Then use the contents of that variable to perform the operations specified within the body of the loop.*

More compact syntax

As you can see, this approach does not require you to get an iterator and to explicitly use that iterator to sequentially access the elements in the collection. Thus, the syntax is more compact than the syntax shown in Listing 3 . Further, by eliminating the requirement to get the iterator, this construct also eliminates the requirement for you to qualify the code using the angle-bracket syntax. All of those details are handled automatically behind the scenes.

Not quite as powerful as an iterator

Although not shown in Listing 3 , the use of an iterator allows you to remove the most recently accessed element from a collection. As near as I can tell, the enhanced **for** loop does not provide that capability. Therefore, the enhanced **for** loop is not quite as powerful as an iterator. However, if you don't need to remove elements from the collection, the enhanced **for** loop appears to be a good and somewhat simpler alternative to an iterator.

The output

The code in Listing 4 produced the last three lines of text in the output shown in Figure 5 . Obviously the last three lines match the first three lines since they simply represent different ways to produce a text representation of the same three **Date** objects.

Run the programs

I encourage you to copy the code from Listing 5 through Listing 9 Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, I explained:

- How the Java Collections Framework behaved prior to the release of Java version 1.5.
- The effect of an incorrect cast in code that doesn't use generics.

- How to avoid the requirement to cast through the use of generics, including an illustration of some of the required syntax for generics.
- The compile-time type safety provided by the use of generics.
- The syntax requirements for the use of iterators with generics.
- The use of the enhanced **for** loop with collections.

What's next?

Future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

Those modules will teach you about other aspects of generics including:

- Generic methods
- Generic classes
- Raw types in generics
- Wildcards and bounded wildcards in generics
- The impact of inheritance on generics

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Getting Started with Generics
- File: Java4210.htm
- Published: 09/01/2013

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are provided below.

Note:
Listing 5 - The program named Generics01.

```
/*File Generics01.java
Copyright 2005, R.G.Baldwin

Illustrates requirement to cast without the use
of generics.

V1.5 compiler produces following warning:

Note: Generics01.java uses unchecked or unsafe
operations.
```

```
Note: Recompile with -Xlint:unchecked for
details.

Recompilation with -Xlint:unchecked produces
the following output:

Generics01.java:34: warning: [unchecked] unchecked
call to add(E) as a member of
 the raw type ArrayList
    var1.add(new Date());
             ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
1 warning

Tested using JDK 1.7 under Win 7.
**********************************************/

import java.util.*;

public class Generics01{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  ((Date)var1.get(0)).getTime());
  }//runIt

  public static void main(String[] args){
    new Generics01().runIt();
  }//end main

}//end class Generics01
//=============================================//
```

Note:

Listing 6 - The program named Generics02.

```
/*File Generics02.java
Copyright 2005, R.G.Baldwin

Illustrates the application of an incorrect cast
to an element that is fetched from an ArrayList
object and the runtime error produced by that
incorrect cast.

This program does not produce a compiler error,
although the v1.5 compiler does produce a
general warning having to do with the failure
to apply the new generics syntax released in
v1.5. (Earlier compilers would not have produced
such a warning.)

However, the compiler does not check to confirm
that the correct cast is applied.  This results
in the following runtime error when an incorrect
cast is applied:

Exception in thread "main"
java.lang.ClassCastException: java.lang.String
cannot
 be cast to java.util.Date
        at Generics02.runIt(Generics02.java:46)
        at Generics02.main(Generics02.java:53)


Tested using JDK 1.7 under Win 7.
**********************************************/
```

```
import java.util.*;

public class Generics02{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    var1.add("abcd");
    //Note that the (Date) cast is applied to
    // an element of type String in the following
    // statement, producing a runtime error.  The
    // problem is that the wrong index was used
    // in fetching the element.  Thus, the wrong
    // element was fetched.
    System.out.println(
                  ((Date)var1.get(1)).getTime());
    System.out.println(
                 ((String)var1.get(1)).length());
  }//end runIt

  public static void main(String[] args){
    new Generics02().runIt();
  }//end main

}//end class Generics02
//===========================================//
```

Listing 7 - The program named Generics03.

Listing 7 - The program named Generics03.

```
/*File Generics03.java
Copyright 2005, R.G.Baldwin

Illustrates use of generics to avoid requirement
to cast.  Requires v1.5 or later.

Program output for one run was:

1377995720768

Tested using JDK 1.7 under Win 7.
**********************************************/

import java.util.*;

public class Generics03{

  ArrayList <Date> var1 = new ArrayList<Date>();

  void runIt(){
    var1.add(new Date());
    //Note that no cast is required in the
    // following statement.
    System.out.println(var1.get(0).getTime());
  }//end runIt

  public static void main(String[] args){
    new Generics03().runIt();
  }//end main

}//end class Generics03
//============================================//
```

Listing 8 - The program named Generics04.

```
/*File Generics04.java
Copyright 2005, R.G.Baldwin

Illustrates ability of generics to prevent
storing of wrong type in a collection.
Requires v1.5 or later.

Compilation produces following error message:

Generics04.java:34: error: no suitable method
found for add(String)
    var1.add("abcd");
        ^
    method ArrayList.add(int,Date) is not
    applicable (actual and formal argument lists
    differ in length)
    method ArrayList.add(Date) is not applicable
      (actual argument String cannot be converted
      to Date by method invocation conversion)
1 error

Once the ArrayList has been declared to be of
type Date, it is not possible to add an element
of type String.  An attempt to do so produces
a compiler error.

Tested using JDK 1.7 under Win 7.
**********************************************/

import java.util.*;

public class Generics04{

  ArrayList <Date> var1 = new ArrayList<Date>();
```

Listing 8 - The program named Generics04.

```
  void runIt(){
    var1.add("abcd");
    System.out.println(var1.get(0).getTime());
  }//end runIt

  public static void main(String[] args){
    new Generics04().runIt();
  }//end main

}//end class Generics04
//=============================================//
```

Note:

Listing 9 - The program named Generics05.

```
/*File Generics05.java
Copyright 2005, R.G.Baldwin

Illustrates required syntax for using an iterator
with generics.  Also illustrates the new for-each
construct in Java 5.0

Output for one particular run is shown below.

1378070280877
1378156680877
1378243080877

1378070280877
1378156680877
```

```
1378243080877

Output will be different each time the program
is run depending on the current date and time.

Tested using JDK 1.7 under Win 7.
**********************************************/

import java.util.*;

public class Generics05{
  //Create an ArrayList object suitable for
  // storing references to Date objects.
  ArrayList <Date> var1 = new ArrayList<Date>();

  void runIt(){
    //Get current date and time in milliseconds.
    long now = new Date().getTime();
    //Get length of one day in milliseconds
    long oneDay = 24 * 60 * 60 * 1000;

    //Populate the ArrayList object
    var1.add(new Date(now));
    var1.add(new Date(now + oneDay));
    var1.add(new Date(now + 2 * oneDay));

    //Get an iterator
    Iterator <Date> iter = var1.iterator();

    //Perform the iteration
    while(iter.hasNext()){
      System.out.println(iter.next().getTime());

    }//end while loop

    System.out.println();//blank line

    //Now perform the same iteration using
```

```
    // the new for-each construct.
    for(Date element : var1){
      System.out.println(element.getTime());

    }//end for-each

  }//end runIt

  public static void main(String[] args){
    new Generics05().runIt();
  }//end main

}//end class Generics05
//=============================================//
```

-end-

Java4210r: Review of Getting Started
This module contains review questions and answers keyed to the module titled Java4210: Getting Started with Generics

Revised: Wed May 11 17:31:29 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
    - Question 7
    - Question 8
    - Question 9
    - Question 10
    - Question 11
    - Question 12
    - Question 13
- Figures
- Listings
- Answers
    - Answer 13
    - Answer 12
    - Answer 11
    - Answer 10

    - Answer 9
    - Answer 8
    - Answer 7
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1

- Miscellaneous

Preface

This module contains review questions and answers keyed to the module titled Java4210: Getting Started with Generics .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: When Java version 1.5 was released, it contained many new language features, including:

- Generics
- Enhanced for Loop
- Autoboxing/Unboxing
- Collections
- Typesafe Enums
- Varargs
- Static Import
- Metadata

Answer 1

Question 2

True or False: Using Java version1.7, the code shown in Listing 1 will compile without warnings, errors, or other messages being generated by the compiler.

Listing 1 - Question 2

Listing 1 - Question 2

```
/*File Q02.java
**********************************************/

import java.util.*;

public class Q02{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  ((Date)var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new Q02().runIt();
  }//end main

}//end class Q02
//==============================================//
```

Answer 2

Question 3

Generics was introduced in Java v1.5 to provide which of the following:

- A. Code simplicity
- B. Compile-time type safety
- C. A clear and straightforward code syntax.
- D. None of the above.

Answer 3

Question 4

True or False: The use of generics is limited to the Java Collections framework.

Answer 4

Question 5

True or False: The Java v1.7 compiler produces an error message when an attempt is made to compile the code shown in Listing 1 .

Answer 5

Question 6

True or False: When the code in Listing 1 is compiled using the -**Xlint:unchecked** switch with Java v1.7, the compiler produces:

- A. An error message
- B. A warning
- C. None of the above

Answer 6

Question 7

True or False: The program code shown in Listing 2 will compile and execute successfully to produce an output similar to that shown in Figure 1 .

Listing 2 - Question 7.

```
/*File Q07.java
**********************************************/

import java.util.*;

public class Q07{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    System.out.println(
                  (var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new Q07().runIt();
  }//end main

}//end class Q07
//==============================================//
```

Figure 1 - Question 7.

```
1378054267172
```

Answer 7

Question 8

What modification must be made to the code shown in Listing 2 to cause it to compile and execute successfully?

Answer 8

Question 9

True or False: The program code shown in Listing 3 will compile and execute successfully to produce an output similar to that shown in Figure 2 .

Listing 3 - Question 9.

Listing 3 - Question 9.

```
/*File Q09.java
**********************************************/

import java.util.*;

public class Q09{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    var1.add("abcd");
    System.out.println(
                  ((Date)var1.get(1)).getTime());
    System.out.println(
                 ((String)var1.get(1)).length());
  }//end runIt

  public static void main(String[] args){
    new Q09().runIt();
  }//end main

}//end class Q09
//==============================================//
```

Figure 2 - Question 9.

Figure 2 - Question 9.

```
1378056276557
4
```

Answer 9

Question 10

The code shown in Listing 4 will not compile and run successfully. Without using a cast operator, what changes can you make to the class named Q10 to cause the program to compile and run successfully and to produce the current time in milliseconds?

Listing 4 - Question 10.

Listing 4 - Question 10.

```
/*File Q10.java
**********************************************/

import java.util.*;

public class Q10{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  (var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new Q10().runIt();
  }//end main

}//end class Q10
//==============================================//
```

Answer 10

Question 11

True or False: Using Java v1.7, even with the empty angle brackets shown in the instantiation of the ArrayList object in Listing 5 , the program in Listing 5

will compile and run successfully with no notes, warnings, errors, or other complaints from the compiler. The program will produce an output similar to that shown in Figure 3 .

Listing 5 - Question 11.

```
/*File Q11.java
**********************************************/

import java.util.*;

public class Q11{

  ArrayList <Date> var1 = new ArrayList<>();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  (var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new Q11().runIt();
  }//end main

}//end class Q11
//==============================================//
```

Figure 3 - Question 11.

```
1378059224125
```

Answer 11

Question 12

True or False: The program shown in Listing 6 will compile and run successfully producing an output similar to that shown in Figure 4 .

Listing 6 - Question 12.

Listing 6 - Question 12.

```
/*File Q12.java
**********************************************/

import java.util.*;

public class Q12{

  ArrayList <Date> var1 = new ArrayList<Date>();

  void runIt(){
    var1.add("abcd");

    System.out.println(var1.get(0).getTime());
  }//end runIt

  public static void main(String[] args){
    new Q12().runIt();
  }//end main

}//end class Q12
//==============================================//
```

Figure 4 - Question 12.

Figure 4 - Question 12.

```
1378060513651
```

Answer 12

Question 13

True or False: The code shown in Listing 7 will compile and run successfully and produce the output shown in Figure 5 .

Listing 7 - Question 13.

Listing 7 - Question 13.

```
/*File Q13.java
**********************************************/

import java.util.*;

public class Q13{
  ArrayList <Integer> var1 = new
ArrayList<Integer>();

  void runIt(){

    var1.add(new Integer(1));
    var1.add(new Integer(2));
    var1.add(new Integer(3));

    //Get an iterator
    Iterator iter = var1.iterator();

    //Perform the iteration
    while(iter.hasNext()){
      System.out.println(iter.next().intValue());
    }//end while loop

  }//end runIt

  public static void main(String[] args){
    new Q13().runIt();
  }//end main

}//end class Q13
//==============================================//
```

Figure 5 - Question 13.

```
1
2
3
```

Answer 13

Figures

- Figure 1 . Question 7
- Figure 2 . Question 9.
- Figure 3 . Question 11.
- Figure 4 . Question 12.
- Figure 5 . Question 13.
- Figure 6 . Answer 13.
- Figure 7 . Answer 12.
- Figure 8 . Answer 10.
- Figure 9 . Answer 9.
- Figure 10 . Answer 7.
- Figure 11 . Answer 6.
- Figure 12 . Answer 2.

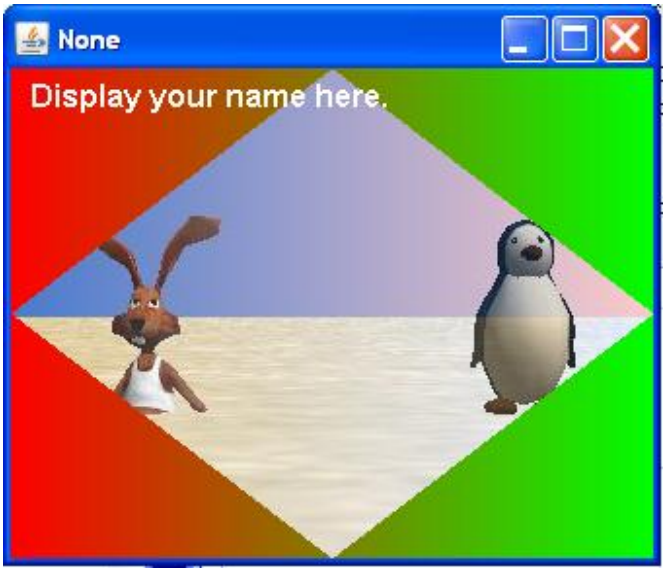
Listings

- Listing 1 . Question 2.
- Listing 2 . Question 7.
- Listing 3 . Question 9.
- Listing 4 . Question 10.
- Listing 5 . Question 11.
- Listing 6 . Question 12.
- Listing 7 . Question 13.
- Listing 8 . Answer 13.
- Listing 9 . Answer 10.

- Listing 10 . Answer 8.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 13

False. This program produces the error shown in Figure 6 below. You must qualify the iterator as shown in Listing 8 to correct the error. If you don't qualify the iterator, the reference is retrieved from the list as type Object. The Object class neither defines nor inherits a method named intValue.

Listing 8 - Answer 13.

Listing 8 - Answer 13.

```
/*File A13.java
**********************************************/

import java.util.*;

public class A13{
  ArrayList <Integer> var1 = new
ArrayList<Integer>();

  void runIt(){

    var1.add(new Integer(1));
    var1.add(new Integer(2));
    var1.add(new Integer(3));

    //Get an iterator
    Iterator <Integer> iter = var1.iterator();

    //Perform the iteration
    while(iter.hasNext()){
      System.out.println(iter.next().intValue());
    }//end while loop

  }//end runIt

  public static void main(String[] args){
    new Q13().runIt();
  }//end main

}//end class A13
//==============================================//
```

Figure 6 - Answer 13.

```
Q13.java:22: error: cannot find symbol

System.out.println(iter.next().intValue());

^
  symbol:   method intValue()
  location: class Object
1 error
```

Back to Question 13

Answer 12

False. This program produces the compiler error shown in Figure 7 below. This is the result of attempting to store an object's reference of type String in a collection reserved exclusively for references to objects of type Date.

Figure 7 - Answer 12.

Figure 7 - Answer 12.

```
Q12.java:13: error: no suitable method found for
add(String)
    var1.add("abcd");
        ^
    method ArrayList.add(int,Date) is not
applicable
      (actual and formal argument lists differ
in length)
    method ArrayList.add(Date) is not applicable
      (actual argument String cannot be
converted to Date by method invocation
      conversion)
1 error
```

Back to Question 12

Answer 11

True. Updates made in Java v1.7 eliminated the requirement to insert the name of a type in those angle brackets in order to make the compiler happy.

Back to Question 11

Answer 10

One solution is to add the generic syntax shown in the class named A10 in Listing 9 below. This will cause the program to compile and execute successfully producing an output similar to that shown in Figure 8 below.

Listing 9 - Answer 10.

```
/*File A10.java
**********************************************/

import java.util.*;

public class A10{

  ArrayList <Date> var1 = new ArrayList<Date>();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  (var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new A10().runIt();
  }//end main

}//end class A10
//==============================================//
```

Figure 8 - Answer 10.

Figure 8 - Answer 10.

```
1378058466149
```

Back to Question 10

Answer 9

False: The program produces the runtime error shown in Figure 9 below. The (Date) cast is applied to the contents of the element at index 1. However, that element contains a reference to an object of type String.

Figure 9 - Answer 9.

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
cannot
 be cast to java.util.Date
        at Q09.runIt(Q09.java:15)
        at Q09.main(Q09.java:22)
```

Back to Question 9

Answer 8

The program must be updated to include the cast operator (Date) shown in Listing 10 below.

Listing 10 - Answer 8.

Listing 10 - Answer 8.

```
/*File Q08.java
**********************************************/

import java.util.*;

public class Q08{

  ArrayList var1 = new ArrayList();

  void runIt(){
    var1.add(new Date());
    //Note the required cast in the following
    // statement.
    System.out.println(
                  ((Date)var1.get(0)).getTime());

  }//runIt

  public static void main(String[] args){
    new Q08().runIt();
  }//end main

}//end class Q08
//==============================================//
```

Back to Question 8

Answer 7

False: This program produces the compiler error shown in Figure 10 below.

Figure 10 - Answer 7.

```
Q07.java:17: error: cannot find symbol
                  (var1.get(0)).getTime());
                                ^
  symbol:   method getTime()
  location: class Object
Note: Q07.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for
details.
1 error
```

Back to Question 7

Answer 6

B. A warning

The compiler output is shown in Figure 11 below:

Figure 11 - Answer 6.

Figure 11 - Answer 6.

```
Q02.java:13: warning: [unchecked] unchecked call
to add(E) as a member of the raw
type ArrayList
    var1.add(new Date());
             ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
1 warning
```

Back to Question 6

Answer 5

False. Two notes are displayed. However, the messages produced by the Java v1.7 compiler in this cases are neither error messages nor warnings. Instead they are simply notes. The notes provide instructions on how to recompile and get more information regarding a potential problem.

Back to Question 5

Answer 4

False. The tentacles of generics reach into many different areas of Java in very complex ways including:

- Generic methods
- Generic classes
- Raw types
- Wildcards
- Generics and inheritance

Back to Question 4

Answer 3

B. Compile-time type safety

Back to Question 3

Answer 2

False: An attempt to compile the program using the Java v1.7 compiler produces an output similar to that shown in Figure 12 below.

Figure 12 - Answer 2.

```
Note: Q02.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for
details.
```

Back to Question 2

Answer 1

False. The Collections framework was not new in Java v1.5. It was available long before version 1.5 was released.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4210r Review
- File: Java4210r.htm
- Published: 09/01/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4220r: Review of Generic Types
This module contains review questions and answers related to generic types in Java.

Revised: Wed May 11 17:44:56 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
    - Question 7
    - Question 8
    - Question 9
    - Question 10
    - Question 11
    - Question 12
    - Question 13
- Figures
- Listings
- Answers
    - Answer 13
    - Answer 12
    - Answer 11
    - Answer 10

    - Answer 9
    - Answer 8
    - Answer 7
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1

- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at:

- Generics (Updated)
- Why Use Generics
- Generic Types

    - Raw Types

This module contains review questions and answers keyed to the material in the above list.

Once you study that material, you should be able to answer the review questions in this module.

Once you complete your study of all the material on generics , you should be able to answer the review questions at Questions and Exercises on the Oracle

site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: Generics add stability to your code by making more of your bugs detectable at runtime.

Answer 1

Question 2

True or False: Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Answer 2

Question 3

True or False: Type parameters provide a way for you to re-use the same code with different inputs. The inputs to type parameters are values.

Answer 3

Question 4

True or False: Code that uses generics provides the following benefits over non-generic code:

- Stronger type checks at compile time.
- Elimination of casts.
- Enabling programmers to implement generic algorithms.

Answer 4

Question 5

True or False: A *generic type* is a generic class or interface that is parameterized over types.

Answer 5

Question 6

True or False: The code shown in Listing 1 will compile and run successfully producing the output shown in Figure 1 .

Listing 1 - Question 6.

Listing 1 - Question 6.

```
/*File Q06.java
**********************************************/
public class Q06{

  public static void main(String[] args){
    Box box = new Box();
    box.set(new Integer(5));
    System.out.println(
              ((Integer)(box.get())).intValue());
  }//end main
}//end class Q06

class Box{
  private Object object;

  public void set(Object object){
        this.object = object;
  }//end set

  public Object get(){
        return object;
  }//end get
}//end Box
//=============================================//
```

Figure 1 - Question 6..

Figure 1 - Question 6..

```
5
```

Answer 6

Question 7

True or False: The code shown in Listing 2 will compile and run successfully producing the output shown in Figure 2 .

Listing 2 - Question 7.

Listing 2 - Question 7.

```
/*File Q07.java
**********************************************/
public class Q07{

  public static void main(String[] args){
    Box <Integer> box = new Box <Integer>();
    box.set(new Integer(5));
    System.out.println(box.get().intValue());
  }//end main
}//end class Q07

class Box{
  private Object object;

  public void set(Object object){
        this.object = object;
  }//end set

  public Object get(){
        return object;
  }//end get
}//end Box
//============================================//
```

Figure 2 - Question 7.

Figure 2 - Question 7.

```
5
```

Answer 7

Question 8

Design and write a class named A08Box which, when combined with the class defined in Listing 3 will compile, run, and produce the output shown in Figure 3 .

Listing 3 - Question 8.

Listing 3 - Question 8.

```
/*File Q08.java
*********************************************/
public class Q08{

  public static void main(String[] args){
    A08Box <Integer> box = new A08Box <Integer>
();
    box.set(new Integer(5));
    System.out.println(box.get().intValue());
  }//end main
}//end class Q08
//=============================================//
```

Figure 3 - Question 8.

```
5
```

Answer 8

Question 9

True or False: The program with the "diamond" syntax shown in Listing 4 will compile and run with no compiler complaints under Java v1.7, producing

the output shown in Figure 4 .

Listing 4 - Question 9.

```
/*File Q09.java
**********************************************/
public class Q09{

  public static void main(String[] args){
    Box <> box = new Box <Integer>();
    box.set(new Integer(5));
    System.out.println(box.get().intValue());
  }//end main
}//end class Q09
//=============================================//
class Box<T>{
  private T t;

  public void set(T t){
        this.t = t;
  }//end set

  public T get(){
        return t;
  }//end get
}//end Box
//=============================================//
```

Figure 4 - Question 9.

```
5
```

Answer 9

Question 10

True or False: The program with the "diamond" syntax shown in Listing 5 will compile and run with no compiler complaints under Java v1.7, producing the output shown in Figure 5 .

Listing 5 - Question 10.

Listing 5 - Question 10.

```
/*File Q10.java
**********************************************/
public class Q10{

  public static void main(String[] args){
    Box <Integer> box = new Box <>();
    box.set(new Integer(5));
    System.out.println(box.get().intValue());
  }//end main
}//end class Q10
//=============================================//
class Box<T>{
  private T t;

  public void set(T t){
        this.t = t;
  }//end set

  public T get(){
        return t;
  }//end get
}//end Box
//=============================================//
```

Figure 5 - Question 10.

Figure 5 - Question 10.

```
5
```

Answer 10

Question 11

Design and write a class named A11Box which, when used in conjunction with the code shown in Listing 6 , will compile and run successfully producing the output shown in Figure 6 .

Listing 6 - Question 11.

```
/*File Q11.java
*********************************************/
public class Q11{

  public static void main(String[] args){
    A11Box <Integer> box =
                    new A11Box <>(new Integer(5));
    System.out.println(box.get().intValue());
  }//end main
}//end class Q11
//=============================================//
```

Figure 6 - Question 11.

```
5
```

Answer 11

Question 12

Design and write a class named A12Box which, when used in conjunction with the code shown in Listing 7 , will compile and run successfully producing the output shown in Figure 7 .

Listing 7 - Question 12.

Listing 7 - Question 12.

```
/*File Q12.java
************************************************/
public class Q12{

  public static void main(String[] args){
    A12Box <String,Integer> boxA =
        new A12Box <>("abcde",new Integer(500));
    System.out.println(boxA.get1());
    System.out.println(boxA.get2());

    A12Box <Integer,String> boxB =
       new A12Box <>(new Integer(900),"fghijkl");
    System.out.println(boxB.get1());
    System.out.println(boxB.get2());
  }//end main
}//end class Q12
//================================================//
```

Figure 7 - Question 12.

```
abcde
500
900
fghijkl
```

Answer 12

Question 13

Given: When mixing legacy code with generic code, you may encounter messages similar to those shown in Figure 8 .

Figure 8 - Question 13.

```
Note: Example.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for
details.
```

True or False: In such cases, the term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety.

Answer 13

Figures

- Figure 1 . Question 6.
- Figure 2 . Question 7.
- Figure 3 . Question 8.
- Figure 4 . Question 9.
- Figure 5 . Question 10.
- Figure 6 . Question 11.

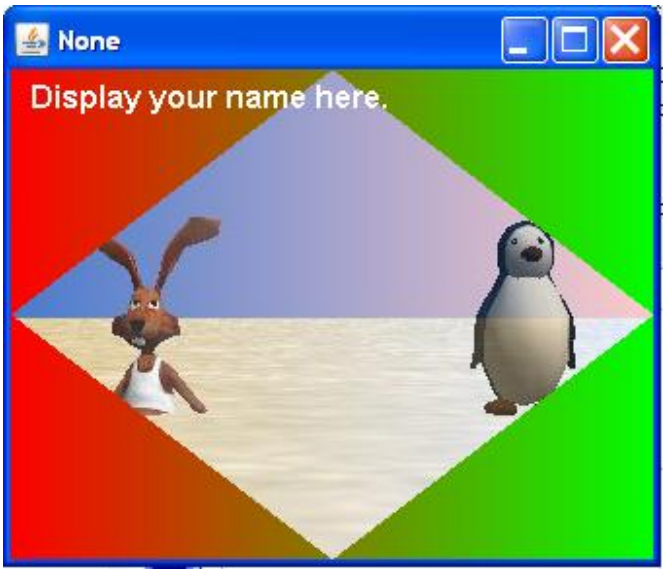
- Figure 7 . Question 12.
- Figure 8 . Question 13.
- Figure 9 . Answer 9.
- Figure 10 . Answer 7.

Listings

- Listing 1 . Question 6.
- Listing 2 . Question 7.
- Listing 3 . Question 8.
- Listing 4 . Question 9.
- Listing 5 . Question 10.
- Listing 6 . Question 11.
- Listing 7 . Question 12.
- Listing 8 . Answer 12.
- Listing 9 . Answer 11.
- Listing 10 . Answer 8.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 13

True.

Back to Question 13

Answer 12

One possible solution is provided in Listing 8 below.

Listing 8 - Answer 12.

Listing 8 - Answer 12.

```
/*File A12Box.java
*************************************************/
public class A12Box<T1,T2>{
  private T1 t1;
  private T2 t2;

  public A12Box(T1 t1,T2 t2){
    this.t1 = t1;
    this.t2 = t2;
  }//end constructor

  public T1 get1(){
    return t1;
  }//end get1

  public T2 get2(){
    return t2;
  }//end get2
}//end A12Box
//=================================================//
```

Back to Question 12

Answer 11

One possible solution is provided in Listing 9 below.

Listing 9 - Answer 11.

```
/*File A11Box.java
************************************************/
public class A11Box<T>{
  private T t;

  public A11Box(T t){
    this.t = t;
  }//end constructor

  public T get(){
    return t;
  }//end get
}//end A11Box
//===============================================//
```

Back to Question 11

Answer 10

True.

Back to Question 10

Answer 9

False. This program produces the error message shown in Figure 9 below.

Figure 9 - Answer 9.

```
Q09.java:8: error: illegal start of type
    Box <> box = new Box <Integer>();
         ^
1 error
```

Back to Question 9

Answer 8

One possible solution is shown in Listing 10 below.

Listing 10 - Answer 8.

Listing 10 - Answer 8.

```
/*File A08Box.java
*********************************************/
public class A08Box<T>{
  private T t;

  public void set(T t){
    this.t = t;
  }//end set

  public T get(){
    return t;
  }//end get
}//end A08Box
//=============================================//
```

Back to Question 8

Answer 7

False. This program produces the compiler errors shown in Figure 10 below.

Figure 10 - Answer 7.

Figure 10 - Answer 7.

```
Q07.java:8: error: type Box does not take
parameters
    Box <Integer> box = new Box <Integer>();
        ^
Q07.java:8: error: type Box does not take
parameters
    Box <Integer> box = new Box <Integer>();
                                ^
2 errors
```

Back to Question 7

Answer 6

True.

Back to Question 6

Answer 5

True.

Back to Question 5

Answer 4

True.

Back to Question 4

Answer 3

False. The correct statement is:

"Type parameters provide a way for you to re-use the same code with different inputs. The inputs to type parameters are **types** ."

Back to Question 3

Answer 2

True,

Back to Question 2

Answer 1

False. The correct statement is

"Generics add stability to your code by making more of your bugs detectable at **compile time** ."

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4220r: Review of Generic Types
- File: Java4220r.htm
- Published: 09/02/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4230r: Review of Generic Methods
This module contains review questions and answers related to generic methods in Java.

Revised: Wed May 11 17:54:19 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
    - Question 7
    - Question 8
- Figures
- Listings
- Answers
    - Answer 8
    - Answer 7
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1

- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at:

- Generic Methods

This module contains review questions and answers keyed to the material in the above document.

Once you study that material, you should be able to answer the review questions in this module.

Once you complete your study of all the material on generics , you should be able to answer the review questions at Questions and Exercises on the Oracle site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: Generic methods are methods that introduce their own type parameters.

Answer 1

Question 2

True or False: Defining a generic method is similar to declaring a generic type. The type parameter's scope is limited to the class in which it is declared.

Answer 2

Question 3

True or False: Generic methods may be either static or non-static.

Answer 3

Question 4

True or False: Generic class constructors are not allowed.

Answer 4

Question 5

Design and write a class named A05Foo which, when combined with the class defined in Listing 1 will compile, run, and produce the output shown in Figure 1 .

Listing 1 - Question 5.

```
/*File Q05.java
**********************************************/
public class Q05{

  public static void main(String[] args){
    A05Foo fooA = new A05Foo();
    System.out.println(fooA.cat("abcd ","efgh"));
    System.out.println(
             fooA.cat("abcd ",new Integer(500)));
    System.out.println(
             fooA.cat(new Integer(500)," abcd"));
  }//end main
}//end class Q05
//==============================================//
```

Figure 1 - Question 5.

```
abcd efgh
abcd 500
500 abcd
```

Answer 5

Question 6

Design and write a class named A06Foo which, when combined with the class defined in Listing 2 will compile, run, and produce the output shown in Figure 2 .

Listing 2 - Question 6.

```
/*File Q06.java
************************************************/
public class Q06{

  public static void main(String[] args){
    A06Foo <String>fooA =   new A06Foo<>();
    fooA.set("_ijklm");
    System.out.println(fooA.cat("abcd ","efgh"));
    System.out.println(
             fooA.cat("abcd ",new Integer(500)));
    System.out.println(
             fooA.cat(new Integer(500)," abcd"));
  }//end main
}//end class Q06
//===============================================//
```

Figure 2 - Question 6.

Figure 2 - Question 6.

```
abcd efgh_ijklm
abcd 500_ijklm
500 abcd_ijklm
```

Answer 6

Question 7

Design and write a class named A07Foo which, when combined with the class defined in Listing 3 will compile, run, and produce the output shown in Figure 3 .

Listing 3 - Question 7.

Listing 3 - Question 7.

```
/*File Q07.java
************************************************/
public class Q07{

  public static void main(String[] args){
    A07Foo <Integer,String>fooA =
                            new A07Foo<>(" ijkl");
    fooA.set(new Integer(-300));
    System.out.println(
                      fooA.cat("abcd "," efgh "));
    System.out.println(
              fooA.cat("abcd ",new Integer(500)));
    System.out.println(
             fooA.cat(new Integer(500)," abcd "));
  }//end main
}//end class Q07
//===============================================//
```

Figure 3 - Question 7.

```
abcd  efgh -300 ijkl
abcd 500-300 ijkl
500 abcd -300 ijkl
```

Answer 7

Question 8

True or False: The syntax for a generic method includes a type parameter, inside angle brackets, and appears after the method's return type

Answer 8

Figures

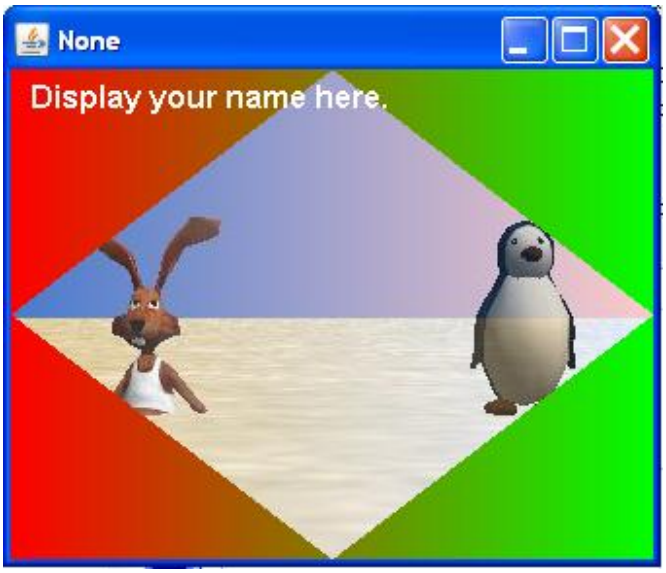
- Figure 1 . Question 5.
- Figure 2 . Question 6.
- Figure 3 . Question 7.

Listings

- Listing 1 . Question 5.
- Listing 2 . Question 6.
- Listing 3 . Question 7.
- Listing 4 . Answer 7.
- Listing 5 . Answer 6.
- Listing 6 . Answer 5.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 8

False. The statement should read:

"The syntax for a generic method includes a type parameter, inside angle brackets, and appears **before** the method's return type"

Back to Question 8

Answer 7

One possible solution is provided in Listing 4 below.

Listing 4 - Answer 7.

Listing 4 - Answer 7.

```
/*File A07Foo.java
************************************************/
//A generic class
public class A07Foo<T3,T4>{
  private T3 tC;
  private T4 tD;

  //constructor
  public A07Foo(T4 tD){
    this.tD = tD;
  }//end constructor

  //A set method
  public void set(T3 tC){
    this.tC = tC;
  }//end cat

  //A generic concatenation method
  public <T1,T2> String cat(T1 tA,T2 tB){
    return tA.toString() + tB.toString()
               + tC.toString() + tD.toString();
  }//end cat

}//end A07Foo
//================================================//
```

Back to Question 7

Answer 6

One possible solution is provided in Listing 5 below.

Listing 5 - Answer 6.

```
/*File A06Foo.java
*********************************************/
public class A06Foo<T3>{
  private T3 tC;

  //A set method
  public void set(T3 tC){
    this.tC = tC;
  }//end cat

  //A generic concatenation method
  public <T1,T2> String cat(T1 tA,T2 tB){
    return tA.toString() + tB.toString()
                                  + tC.toString();
  }//end cat

}//end A06Foo
//=============================================//
```

Back to Question 6

Answer 5

One possible solution is provided in Listing 6 below.

Listing 6 - Answer 5.

```
/*File A05Foo.java
************************************************/
public class A05Foo{
  //A generic concatenation method
  public <T1,T2> String cat(T1 tA,T2 tB){
    return tA.toString() + tB.toString();
  }//end cat
}//end A05Foo
//===============================================//
```

Back to Question 5

Answer 4

False. Generic class constructors **are** allowed.

Back to Question 4

Answer 3

True.

Back to Question 3

Answer 2

False. The statement should read as follows:

"Defining a generic method is similar to declaring a generic type, but the type parameter's scope is limited to the **method** where it is declared."

Back to Question 2

Answer 1

True.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4230r: Review of Generic Methods
- File: Java4230r.htm
- Published: 09/02/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor

do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4240r: Review of Bounded Type Parameters
This module contains review questions and answers related to bounded type parameters in Java.

Revised: Wed May 11 18:05:10 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
    - Question 7
    - Question 8
    - Question 9
    - Question 10
- Figures
- Listings
- Answers
    - Answer 10
    - Answer 9
    - Answer 8
    - Answer 7
    - Answer 6
    - Answer 5
    - Answer 4

    - Answer 3
    - Answer 2
    - Answer 1

- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at:

- Bounded Type Parameters
    - Generic Methods and Bounded Type Parameters

This module contains review questions and answers keyed to the material in the above listed documents.

Once you study that material, you should be able to answer the review questions in this module.

Once you complete your study of all the material on generics , you should be able to answer the review questions at Questions and Exercises on the Oracle site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: Bounded type parameters are used to restrict the types that can be used as arguments in a parameterized type.

Answer 1

Question 2

True or False: To declare a bounded type parameter, list the type parameter's name, followed by the **extends** or **implements** keyword, followed by its upper bound class or interface.

Answer 2

Question 3

True or False: A type parameter can have multiple bounds as shown in Figure 1 .

Figure 1 - Question 3.

```
<T extends B1 & B2 & B3>
```

Answer 3

Question 4

True or False: The code shown in Listing 1 will compile and run successfully producing the output shown in Figure 2 .

Listing 1 - Question 4.

```
/*File Q04.java
*********************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q04{

  public static void main(String[] args){
    Foo <Window> aFoo = new Foo <>();
    aFoo.runIt();
  }//end main
}//end class Q04
//=============================================//
class Foo<T extends Frame>{
  public void runIt(){
    System.out.println("Object instantiated.");
  }//end runIt
}//end Foo
```

Figure 2 - Question 4.

```
Object instantiated.
```

Answer 4

Question 5

Define and write a class named A05Foo which, when combined with the code shown in Listing 2 will produce the output shown in Figure 3 .

Listing 2 - Question 5.

Listing 2 - Question 5.

```
/*File Q05.java
**********************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q05{

  public static void main(String[] args){
    A05Foo <Window> aA05Foo = new A05Foo <>();
    aA05Foo.runIt();
  }//end main
}//end class Q05
//==============================================//
```

Figure 3 - Question 5.

```
Object instantiated.
```

Answer 5

Question 6

Define and write a class named A06Foo which, when combined with the code shown in Listing 3 will produce the output shown in Figure 4 .

Listing 3 - Question 6.

```
/*File Q06.java
*************************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q06{

  public static void main(String[] args){
    A06Foo <Window> aA06Foo =
          new A06Foo <Window>(new Frame());
    aA06Foo.runIt();
  }//end main
}//end class Q06
//=============================================//
```

Figure 4 - Question 6.

```
Object instantiated.
class java.awt.Frame
```

Answer 6

Question 7

True or False: The code shown in Listing 4 will compile and run successfully producing the output shown in Figure 5 .

Listing 4 - Question 7.

Listing 4 - Question 7.

```
/*File Q07.java
************************************************/
import java.awt.*;
import javax.swing.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
*/
public class Q07{
  public static void main(String[] args){
    JFrame aFrame = new JFrame("This is a
title");
    System.out.println(aFrame.getTitle());

    Foo  aFoo = new Foo();
    aFoo.runIt(aFrame);
  }//end main
}//end class Q07
//===============================================//

class Foo{
  public <T extends Frame> void runIt(T ref){
    System.out.println("Running runIt");
    System.out.println(ref.getTitle());
  }//end runIt
}//end Foo

//===============================================//
```

Figure 5 - Question 7.

```
This is a title
Running runIt
This is a title
```

Answer 7

Question 8

True or False: The code shown in Listing 5 will compile and run successfully producing the output shown in Figure 6 .

Listing 5 - Question 8.

Listing 5 - Question 8.

```
/*File Q08.java
**********************************************/
import java.awt.*;
import javax.swing.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
*/
public class Q08{
  public static void main(String[] args){
    Frame aFrame = new Frame("This is a title");
    System.out.println(aFrame.getTitle());

    Foo  aFoo = new Foo();
    aFoo.runIt(aFrame);
  }//end main
}//end class Q08
//=============================================//

class Foo{
  public <T extends JFrame> void runIt(T ref){
    System.out.println("Running runIt");
    System.out.println(ref.getTitle());
  }//end runIt
}//end Foo
```

Figure 6 - Question 8.

```
This is a title
```

Answer 8

Question 9

True or False: The code shown in Listing 6 will compile and run successfully producing the output shown in Figure 7 .

Listing 6 - Question 9.

Listing 6 - Question 9.

```
/*File Q09.java
************************************************/
import java.awt.*;
import javax.swing.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
*/
public class Q09{
  public static void main(String[] args){
    JFrame aFrame = new JFrame("This is a
title");
    System.out.println(aFrame.getTitle());

    Foo  aFoo = new Foo();
    aFoo.runIt(aFrame);
  }//end main
}//end class Q09
//=============================================//

class Foo{
  public <T> void runIt(T ref){
    System.out.println("Running runIt");
    System.out.println(ref.getTitle());
  }//end runIt
}//end Foo

//=============================================//
```

Figure 7 - Question 9.

```
This is a title
```

Answer 9

Question 10

True or False: The code shown in Listing 7 will compile and run successfully producing the output shown in Figure 8 .

Listing 7 - Question 10.

Listing 7 - Question 10.

```
/*File Q10.java
**************************************************/
import java.awt.*;
import javax.swing.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
*/
public class Q10{
  public static void main(String[] args){
    JFrame aFrame = new JFrame("This is a
title");
    System.out.println(aFrame.getTitle());

    Foo  aFoo = new Foo();
    aFoo.runIt(aFrame);
  }//end main
}//end class Q10
//================================================//

class Foo{
  public <T extends Object> void runIt(T ref){
    System.out.println("Running runIt");
    System.out.println(ref.getTitle());
  }//end runIt
}//end Foo

//================================================//
```

Figure 8 - Question 10.

```
This is a title
```

Answer 10

Figures

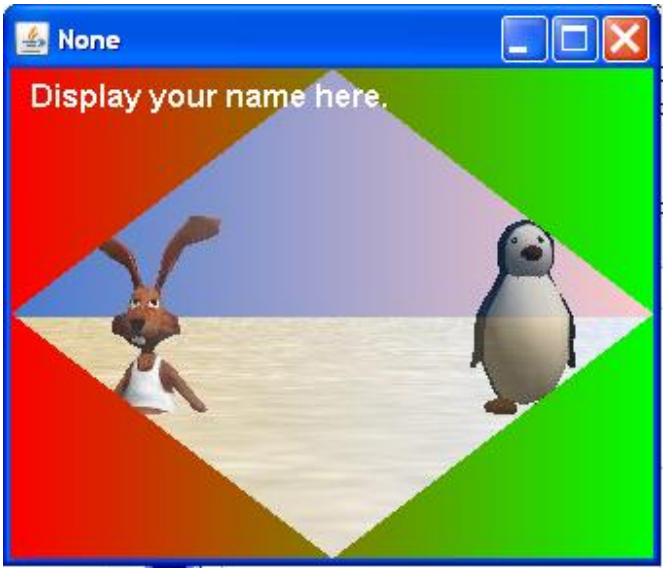
- Figure 1 . Question 3.
- Figure 2 . Question 4.
- Figure 3 . Question 5.
- Figure 4 . Question 6.
- Figure 5 . Question 7.
- Figure 6 . Question 8.
- Figure 7 . Question 9.
- Figure 8 . Question 10.
- Figure 9 . Answer 10.
- Figure 10 . Answer 9.
- Figure 11 . Answer 8.
- Figure 12 . Answer 4.

Listings

- Listing 1 . Question 4.
- Listing 2 . Question 5.
- Listing 3 . Question 6.
- Listing 4 . Question 7.
- Listing 5 . Question 8.
- Listing 6 . Question 9.
- Listing 7 . Question 10.
- Listing 8 . Answer 6.
- Listing 9 . Answer 5.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 10

False. This is an unsuccessful call to generic method with specified upper-bound of Object. The incoming reference is received as type Object and must be downcast to at least Frame to call the method named getTitle. The incoming parameter is received as the type of the specified upper-bound. This results in the compiler error shown in Figure 9 .

Figure 9 - Answer 10.

```
Q10.java:36: error: cannot find symbol
    System.out.println(ref.getTitle());
                       ^
  symbol:   method getTitle()
  location: variable ref of type T
  where T is a type-variable:
    T extends Object declared in method
<T>runIt(T)
1 error
```

Back to Question 10

Answer 9

False. This is an unsuccessful call to generic method with default upper-bound of Object. The incoming reference is received as type Object and must be downcast to at least Frame to call the method named getTitle. The incoming parameter is received as the type of the specified upper-bound which is Object by default. This results in the compiler error shown in Figure 10 .

Figure 10 - Answer 9.

```
Q09.java:36: error: cannot find symbol
    System.out.println(ref.getTitle());
                           ^
  symbol:   method getTitle()
  location: variable ref of type T
  where T is a type-variable:
    T extends Object declared in method
<T>runIt(T)
1 error
```

Back to Question 9

Answer 8

False. This is an unsuccessful call to a generic method where the type of the object passed to the method is above the upper bound on the specified generic type for the method. An attempt to compile the program results in the error shown in Figure 11 below.

Figure 11 - Answer 8.

Figure 11 - Answer 8.

```
Q08.java:25: error: method runIt in class Foo
cannot be applied to given types;
    aFoo.runIt(aFrame);
        ^
  required: T
  found: Frame
  reason: inferred type does not conform to
declared bound(s)
    inferred: Frame
    bound(s): JFrame
  where T is a type-variable:
    T extends JFrame declared in method
<T>runIt(T)
1 error
```

Back to Question 8

Answer 7

True. This is a successful call to a generic method where the generic type for the method is above *(in the inheritance hierarchy)* the type of object passed as a parameter.

Back to Question 7

Answer 6

One possible solution is shown in Listing 8 below.

Listing 8 - Answer 6.

```
/*File A06Foo.java
**************************************************/
import java.awt.*;

public class A06Foo<T extends Component>{
  private T param;

  public A06Foo(T param){
    this.param = param;
  }//end constructor

  public void runIt(){
    System.out.println("Object instantiated.");
    System.out.println(param.getClass());
  }//end runIt
}//end A06Foo
//================================================//
```

Back to Question 6

Answer 5

One possible solution is shown in Listing 9 below where **T extends Component** . However, causing T to extend Window, Container, Component, or Object would all be valid solutions. The upper bound for T must be at or above the type being passed as a type parameter in the inheritance hierarchy.

Listing 9 - Answer 5.

```
/*File A05Foo.java
**********************************************/
import java.awt.*;

public class A05Foo<T extends Component>{
  public void runIt(){
    System.out.println("Object instantiated.");
  }//end runIt
}//end A05Foo
//=============================================//
```

Back to Question 5

Answer 4

False. This program produces the compiler error shown in Figure 12 below. The problem is that the upper bound for the type parameter in the class named Foo (Frame) is below the type passed as a parameter (Window) in the attempt to instantiate an object of the class named Foo.

Figure 12 - Answer 4.

Figure 12 - Answer 4.

```
Q04.java:20: error: type argument Window is not
within bounds
of type-variable T
    Foo <Window> aFoo = new Foo <>();
         ^
  where T is a type-variable:
    T extends Frame declared in class Foo
Q04.java:20: error: cannot infer type arguments
for Foo<>;
    Foo <Window> aFoo = new Foo <>();
                                ^
  reason: no instance(s) of type variable(s) T
exist so that
  Foo<T> conforms to
Foo<Window>
  where T is a type-variable:
    T extends Frame declared in class Foo
2 errors
```

Back to Question 4

Answer 3

True. A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first.

Back to Question 3

Answer 2

False. The correct statement is:

"To declare a bounded type parameter, list the type parameter's name, followed by the **extends** keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces)."

Back to Question 2

Answer 1

True. There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4240r: Review of Bounded Type Parameters
- File: Java4240r.htm
- Published: 10/19/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be

aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4250r: Review of Generics, Inheritance, and Subtypes
This module contains review questions and answers related to generics, inheritance, and subtypes in Java.

Revised: Wed May 11 18:13:42 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
- Figures
- Listings
- Answers
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1
- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at Generics, Inheritance, and Subtypes .

This module contains review questions and answers keyed to the material in that document.

Once you study that material, you should be able to answer the review questions in this module.

Once you complete your study of all the material on generics , you should be able to answer the review questions at Questions and Exercises on the Oracle site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: The code shown in Listing 1 will compile and run successfully producing the output shown in Figure 1 .

Listing 1 - Question 1.

Listing 1 - Question 1.

```
/*File Q01.java
**********************************************/
/*
Integer and Double are both subclasses of
Number. Both classes define a method named
doubleValue that returns the encapsulated
numeric value as type double.
*/
public class Q01{

  public static void main(String[] args){
    Foo <Integer> iFoo = new Foo <Integer>(15);
    System.out.println(iFoo.get().doubleValue());

    Foo <Double> dFoo = new Foo <Double>(1.0/3);
    System.out.println(dFoo.get().doubleValue());

  }//end main
}//end class Q01
//===========================================//

class Foo<T extends Number>{
  private T obj;

  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
}//end Foo
//===========================================//
```

Figure 1 - Question 1.

```
15.0
0.3333333333333333
```

Answer 1

Question 2

True or False: The code shown in Listing 2 will compile and run successfully producing the output shown in Figure 2 .

Listing 2 - Question 2.

```
/*File Q02.java
**********************************************/
import java.awt.Frame;
import javax.swing.JInternalFrame;
import java.awt.Container;
/*
Frame and JInternalFrame are both subclasses of
```

Listing 2 - Question 2.

```
Container. Both classes define a constructor
that accepts a String as a title. Both classes
define a method named getTitle that returns the
title string.
*/
public class Q02{

  public static void main(String[] args){
    Foo <Frame> fFoo =
        new Foo <Frame>(
            new Frame("Frame title"));
    System.out.println(fFoo.get().getTitle());

    Foo <JInternalFrame> jFoo =
        new Foo <JInternalFrame>(
            new JInternalFrame(
                "JInternalFrame title"));
    System.out.println(jFoo.get().getTitle());

  }//end main
}//end class Q02
//===========================================//

class Foo<T extends Container>{
  private T obj;

  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
}//end Foo
//===========================================//
```

Figure 2 - Question 2.

```
Frame title
JInternalFrame title
```

Answer 2

Question 3

True or False: The code shown in Listing 3 will compile and run successfully producing the output shown in Figure 3 .

Listing 3 - Question 3.

```
/*File Q03.java
**********************************************/
public class Q03{
/*
Integer and Double are both subclasses of
Number. Both classes define a method named
doubleValue that returns the encapsulated
numeric value as type double.
*/
  public static void main(String[] args){
    Foo <Integer> iFoo = new Foo <Integer>(15);
    System.out.println(iFoo.get().doubleValue());
```

Listing 3 - Question 3.

```
    Foo <Double> dFoo = new Foo <Double>(1.0/3);
    System.out.println(dFoo.get().doubleValue());

    displayClassI(iFoo);
    displayClassD(dFoo);
  }//end main
  //-----------------------------------------//

  static void displayClassI(Foo<Integer> obj){
    System.out.println(obj.getClass());
  }//end displayClassI
  //-----------------------------------------//

  static void displayClassD(Foo<Double> obj){
    System.out.println(obj.getClass());
  }//end displayClassD
}//end class Q03
//===========================================//

class Foo<T extends Number>{
  private T obj;

  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
}//end Foo
//===========================================//
```

Figure 3 - Question 3.

```
15.0
0.3333333333333333
class Foo
class Foo
```

Answer 3

Question 4

True or False: The code shown in Listing 4 will compile and run successfully producing the output shown in Figure 4 .

Listing 4 - Question 4.

```
/*File Q04.java
************************************************/
import java.awt.Frame;
import javax.swing.JInternalFrame;
import java.awt.Container;
/*
Frame and JInternalFrame are both subclasses of
Container. Both classes define a constructor
that accepts a String as a title. Both classes
define a method named getTitle that returns the
title string.
*/
public class Q04{

  public static void main(String[] args){
    Foo <Frame> fFoo =
```

```
        new Foo <Frame>(
            new Frame("Frame title"));
    System.out.println(fFoo.get().getTitle());

    Foo <JInternalFrame> jFoo =
        new Foo <JInternalFrame>(
            new JInternalFrame(
                "JInternalFrame title"));
    System.out.println(jFoo.get().getTitle());

    displayClassF(fFoo);
    displayClassJ(jFoo);
  }//end main
  //-----------------------------------------//

  static void displayClassF(Foo<Frame> obj){
    System.out.println(obj.getClass());
  }//end displayClassF
  //-----------------------------------------//

  static void displayClassJ(
                        Foo<JInternalFrame> obj){
    System.out.println(obj.getClass());
  }//end displayClassJ
}//end class Q04
//=========================================//

class Foo<T extends Container>{
  private T obj;

  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
```

```
}//end Foo
//===========================================//
```

Figure 4 - Question 4.

```
Frame title
JInternalFrame title
class Foo
class Foo
```

Answer 4

Question 5

True or False: The code shown in Listing 5 will compile and run successfully producing the output shown in Figure 5 .

Listing 5 - Question 5.

```
/*File Q05.java
*********************************************/
/*
```

Listing 5 - Question 5.

```
Integer and Double are both subclasses of
Number. Both classes define a method named
doubleValue that returns the encapsulated
numeric value as type double. Both classes
also contain a method named intValue that
returns the encapsulated numeric value as
type int.
*/
public class Q05{

  public static void main(String[] args){
    Foo <Integer> iFoo = new Foo <Integer>(15);
    System.out.println(iFoo.get().intValue());

    displayClass(iFoo);
  }//end main
  //-----------------------------------------//

  static void displayClass(Foo<Number> obj){
    System.out.println(obj.getClass());
  }//end displayClass
}//end class Q05
//=========================================//

class Foo<T extends Number>{
  private T obj;

  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
}//end Foo
//=========================================//
```

Listing 5 - Question 5.

Figure 5 - Question 5.

```
15
class Foo
```

Answer 5

Question 6

True or False: The code shown in Listing 6 will compile and run successfully producing the output shown in Figure 6 .

Listing 6 - Question 6.

```
/*File Q06.java
**************************************************/
import java.awt.Frame;
import javax.swing.JInternalFrame;
import java.awt.Container;
/*
```

Listing 6 - Question 6.

```
Frame and JInternalFrame are both subclasses of
Container. Both classes define a constructor
that accepts a String as a title. Both classes
define a method named getTitle that returns the
title string.
*/
public class Q06{

  public static void main(String[] args){
    Foo <Frame> fFoo =
        new Foo <Frame>(
            new Frame("Frame title"));
    System.out.println(fFoo.get().getTitle());

    displayClass(fFoo);
  }//end main
  //-----------------------------------------//

  static void displayClass(Foo<Container> obj){
    System.out.println(obj.getClass());
  }//end displayClass

}//end class Q06
//=========================================//

class Foo<T extends Container>{
  private T obj;

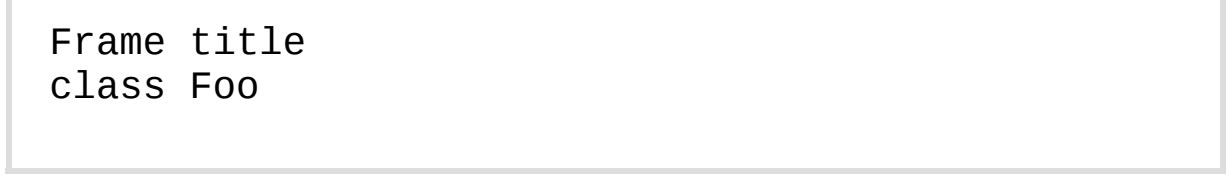
  public Foo(T obj){
    this.obj = obj;
  }//end constructor

  public T get(){
    return obj;
  }//end get
```

Listing 6 - Question 6.

```
}//end Foo
//=============================================//
```

Figure 6 - Question 6.



Answer 6

Figures

- Figure 1 . Question 1.
- Figure 2 . Question 2.
- Figure 3 . Question 3.
- Figure 4 . Question 4.
- Figure 5 . Question 5.
- Figure 6 . Question 6.
- Figure 7 . Answer 6.
- Figure 8 . Answer 5.

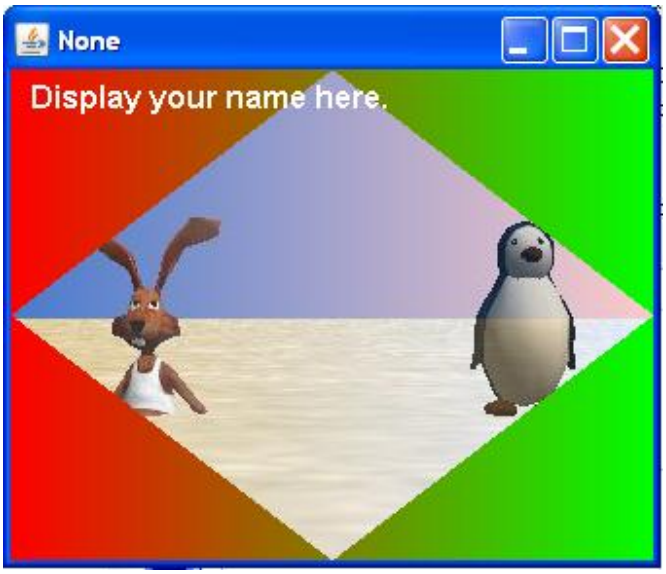
Listings

- Listing 1 . Question 1.
- Listing 2 . Question 2.
- Listing 3 . Question 3.

- Listing 4 . Question 4.
- Listing 5 . Question 5.
- Listing 6 . Question 6.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 6

False. This program illustrates that you cannot pass an object of a parameterized class to a method that is designed to accept an incoming parameter of that class with a specific type of parameter that is a superclass of the actual type of parameter. An attempt to compile the program results in the error shown in Figure 7 .

Figure 7 - Answer 6.

Figure 7 - Answer 6.

```
Q06.java:29: error: method displayClass in class
Q06 cannot be
applied to given types;
    displayClass(fFoo);
    ^
  required: Foo<Container>
  found: Foo<Frame>
  reason: actual argument Foo<Frame> cannot be
converted to
  Foo<Container> by method invocation conversion
1 error
```

Back to Question 6

Answer 5

False. This program illustrates that you cannot pass an object of a parameterized class to a method that is designed to accept an incoming parameter of that class with a specific type of parameter that is a superclass of the actual type of parameter. An attempt to compile the program results in the error shown in Figure 8 .

Figure 8 - Answer 5.

Figure 8 - Answer 5.

```
Q05.java:23: error: method displayClass in class
Q05 cannot be
applied to given types;
    displayClass(iFoo);
    ^
  required: Foo<Number>
  found: Foo<Integer>
  reason: actual argument Foo<Integer> cannot be
converted to
  Foo<Number> by method invocation conversion
1 error
```

Back to Question 5

Answer 4

True. This program illustrates passing an object of a parameterized class to a method that is designed to accept an incoming parameter of that class with a specific type of parameter.

Back to Question 4

Answer 3

True. This program illustrates passing an object of a parameterized class to a method that is designed to accept an incoming parameter of that class with a specific type of parameter.

Back to Question 3

Answer 2

True. This program illustrates the successful use of bounded type parameters.

Back to Question 2

Answer 1

True. This program illustrates the successful use of bounded type parameters. It also illustrates the use of autoboxing to encapsulate primitive types int and double into objects of the wrapper classes Integer and Double.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4250r: Review of Generics, Inheritance, and Subtypes
- File: Java4250r.htm
- Published: 10/20/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4260r: Review of Type Inference
This module contains review questions and answers related to generic type inference in Java.

Revised: Wed May 11 18:23:25 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
- Figures
- Listings
- Answers
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1
- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at Type Inference . This module contains review questions and answers keyed to the material in that document.

Once you study that material, you should be able to answer the review questions in this module.

Once you complete your study of all the material on generics , you should be able to answer the review questions at Questions and Exercises on the Oracle site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Note that much of the type inference material discussed in this module requires the use of Java SE 7 or a later version.

Questions

Question 1 .

True or False: The code shown in Listing 1 will compile and run successfully producing the output shown in Figure 1 .

Listing 1 - Question 1.

```
/*File Q01.java
**************************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q01{

  public static void main(String[] args){
    Foo <Window> var = new Foo <Window>();
    var.runIt();
  }//end main
}//end class Q01
//===============================================//

class Foo<T extends Component>{
  public void runIt(){
    System.out.println("Object instantiated.");
  }//end runIt
}//end Foo
```

Figure 1 - Question 1.

Figure 1 - Question 1.

```
Object instantiated.
```

Answer 1

Question 2

True or False: The code shown in Listing 2 will compile and run successfully producing the output shown in Figure 2 .

Listing 2 - Question 2.

Listing 2 - Question 2.

```
/*File Q02.java
**********************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q02{

  public static void main(String[] args){
    Foo <Window> var = new Foo <>();
    var.runIt();
  }//end main
}//end class Q02
//=============================================//

class Foo<T extends Component>{
  public void runIt(){
    System.out.println("Object instantiated.");
  }//end runIt
}//end Foo
```

Figure 2 - Question 2.

Figure 2 - Question 2.

```
Object instantiated.
```

Answer 2

Question 3

True or False: The code shown in Listing 3 will compile and run successfully producing the output shown in Figure 3 .

Listing 3 - Question 3.

Listing 3 - Question 3.

```
/*File Q03.java
**********************************************/
import java.awt.*;
/*Note the following inheritance hierarchy
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
*/
public class Q03{

  public static void main(String[] args){
    Foo <> var = new Foo <Window>();
    var.runIt();
  }//end main
}//end class Q03
//=============================================//

class Foo<T extends Component>{
  public void runIt(){
    System.out.println("Object instantiated.");
  }//end runIt
}//end Foo
```

Figure 3 - Question 3.

Figure 3 - Question 3.

```
Object instantiated.
```

Answer 3

Question 4

True or False: The code shown in Listing 4 will compile and run successfully producing the output shown in Figure 4 .

Listing 4 - Question 4.

```
/*File Q04.java
**********************************************/
public class Q04{

  public static void main(String[] args){
    Foo <String,Integer> varA =
           new Foo <String,Integer>("abcde",500);
    System.out.print(varA.get1());
    System.out.println("," + varA.get2());

    Foo <Integer,String> varB =
         new Foo <Integer,String>(900,"fghijkl");
    System.out.print(varB.get1());
```

Listing 4 - Question 4.

```
    System.out.println("," + varB.get2());

  }//end main
}//end class Q04
//===========================================//
class Foo<T1,T2>{
  private T1 t1;
  private T2 t2;

  public Foo(T1 t1,T2 t2){
    this.t1 = t1;
    this.t2 = t2;
  }//end constructor

  public T1 get1(){
    return t1;
  }//end get1

  public T2 get2(){
    return t2;
  }//end get2
}//end Foo
```

Figure 4 - Question 4.

```
abcde,500
900,fghijkl
```

Answer 4

Question 5

True or False: The code shown in Listing 5 will compile and run successfully producing the output shown in I mage 5 .

Listing 5 - Question 5.

```
/*File Q05.java
*********************************************/
public class Q05{

  public static void main(String[] args){
    Foo <String,Integer> varA =
                          new Foo <>("abcde",500);
    System.out.print(varA.get1());
    System.out.println("," + varA.get2());

    Foo <Integer,String> varB =
                        new Foo <>(900,"fghijkl");
    System.out.print(varB.get1());
    System.out.println("," + varB.get2());

  }//end main
}//end class Q05
//===============================================//
class Foo<T1,T2>{
  private T1 t1;
  private T2 t2;
```

Listing 5 - Question 5.

```
  public Foo(T1 t1,T2 t2){
    this.t1 = t1;
    this.t2 = t2;
  }//end constructor

  public T1 get1(){
    return t1;
  }//end get1

  public T2 get2(){
    return t2;
  }//end get2
}//end Foo
```

Figure 5 - Question 5.

```
abcde,500
900,fghijkl
```

Answer 5

Question 6

True or False: The code shown in Listing 6 will compile and run successfully producing the output shown in Figure 6 .

Listing 6 - Question 6.

```
/*File Q06.java
**********************************************/
public class Q06{

  public static void main(String[] args){
    Foo <> varA =
           new Foo <String,Integer>("abcde",500);
    System.out.print(varA.get1());
    System.out.println("," + varA.get2());

    Foo <> varB =
         new Foo <Integer,String>(900,"fghijkl");
    System.out.print(varB.get1());
    System.out.println("," + varB.get2());

  }//end main
}//end class Q06
//===============================================//
class Foo<T1,T2>{
  private T1 t1;
  private T2 t2;

  public Foo(T1 t1,T2 t2){
    this.t1 = t1;
    this.t2 = t2;
  }//end constructor
```

Listing 6 - Question 6.

```
  public T1 get1(){
    return t1;
  }//end get1

  public T2 get2(){
    return t2;
  }//end get2
}//end Foo
```

Figure 6 - Question 6.

```
abcde,500
900,fghijkl
```

Answer 6

Figures

- Figure 1 . Question 1.
- Figure 2 . Question 2.
- Figure 3 . Question 3.
- Figure 4 . Question 4.
- Figure 5 . Question 5.
- Figure 6 . Question 6.
- Figure 7 . Answer 6.

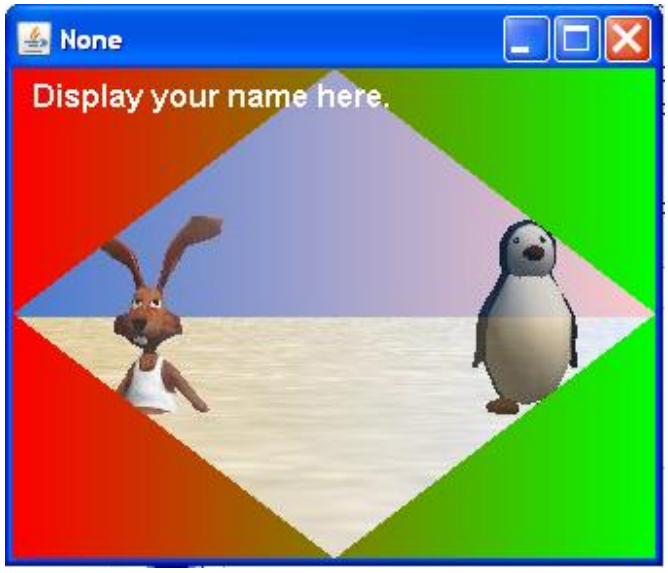
- Figure 8 . Answer 3.

Listings

- Listing 1 . Question 1.
- Listing 2 . Question 2.
- Listing 3 . Question 3.
- Listing 4 . Question 4.
- Listing 5 . Question 5.
- Listing 6 . Question 6.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 6

False. This program illustrates incorrect use of the diamond for type inference and will not compile. In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of matching angle brackets as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, is informally called the diamond. However, in this case, the diamond is in the wrong location. This results in the errors shown in Figure 7 .

Figure 7 - Answer 6.

Figure 7 - Answer 6.

```
Q06.java:18: error: illegal start of type
    Foo <> varA =
         ^
Q06.java:23: error: illegal start of type
    Foo <> varB =
         ^
2 errors
```

Back to Question 6

Answer 5

True. This program illustrates type inference. In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of matching angle brackets as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, is informally called the diamond.

Note that the Foo constructor requires two parameters and the types of those parameters are specified inside the angle brackets.

Also note that the order of the types of the two parameters is swapped between the instantiation of the first and second objects. The program also illustrates autoboxing.

Back to Question 5

Answer 4

True. This program illustrates generics without any obvious type inference. This is the style of programming that would have been used prior to the release of Java SE 7.

Note that the Foo constructor requires two parameters and the types of those parameters are specified inside the angle brackets.

Also note that the order of the types of the two parameters is swapped between the instantiation of the first and second objects. The program also illustrates autoboxing.

Back to Question 4

Answer 3

False. This program illustrates incorrect use of the diamond for type inference and will not compile. In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of matching angle brackets as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, is informally called the diamond. However, in this case, the diamond is in the wrong location. Therefore, the program produces the error shown in Figure 8 .

Figure 8 - Answer 3.

```
Q03.java:26: error: illegal start of type
    Foo <> var = new Foo <Window>();
         ^
1 error
```

Back to Question 3

Answer 2

True. This program illustrates bounded parameters along with type inference. In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of matching angle brackets as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, is informally called the diamond.

Back to Question 2

Answer 1

True. This program illustrates generics with bounded parameters but without any obvious type inference. This is the style of programming that would have been used prior to the release of Java SE 7.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4260r: Review of Type Inference
- File: Java4260r.htm
- Published: 10/22/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4270r: Review of Wildcards
This module contains review questions and answers related to generic wildcards in Java.

Revised: Wed May 11 18:39:23 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Table of Contents
- Preface
- Questions
    - Question 1
    - Question 2
    - Question 3
    - Question 4
    - Question 5
    - Question 6
- Figures
- Listings
- Answers
    - Answer 6
    - Answer 5
    - Answer 4
    - Answer 3
    - Answer 2
    - Answer 1
- Miscellaneous

Preface

This module is one in a collection of modules on Java Generics designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

This and future modules in this series will be based on the Generics (Updated) section of The Java Tutorials from Oracle. *(In the event that these links become broken, you should have no difficulty finding the Oracle material with an online search.)*

You will find the learning resources for this module on the Oracle site at Wildcards .

This module contains review questions and answers keyed to the material in Wildcards .

Once you study that material, you should be able to answer the review questions in this module.

This is the final module in this series. This course does not cover the following topics:

- Type Erasure
- Restrictions on Generics

However, once you complete your study of all the material on generics , *(including the two topics listed above)* , you should be able to answer the review questions at Questions and Exercises on the Oracle site.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: The code shown in Listing 1 will compile and run successfully producing the output shown in Figure 1 .

Note:
Listing 1 - Question 1.

```
/*File Q01.java
**********************************************/
import java.util.ArrayList;
import java.awt.Button;
import java.awt.Label;
import java.awt.Component;

public class Q01{
/*
Given:
Button and Label are subclasses of the Component
class.
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){
    //Create and populate a list designed to
    // store references to objects of type Button
    // and display its contents.
    ArrayList <Button> listA = new ArrayList<>();
    listA.add(new Button("Button01"));
    listA.add(new Button("Button02"));
    display(listA);
    System.out.println("");//blank line
```

```
    //Create and populate a list designed to
    // store references to objects of type Label
    // and display its contents.
    ArrayList <Label> listB = new ArrayList<>();
    listB.add(new Label("Label01"));
    listB.add(new Label("Label02"));
    display(listB);
  }//end main
  //-----------------------------------------//

  static void display(ArrayList<Component> list){
    for(Component n : list){
      System.out.println(n.getClass());
      if(n instanceof Button){
        System.out.println(
                          ((Button)n).getLabel());
      }else if(n instanceof Label) {
        System.out.println(((Label)n).getText());
      }//end else
    }//end for loop
  }//end display
}//end class Q01
```

Figure 1 - Question 1.

Figure 1 - Question 1.

```
class java.awt.Button
Button01
class java.awt.Button
Button02

class java.awt.Label
Label01
class java.awt.Label
Label02
```

Answer 1

Question 2

True or False: The code shown in Listing 2 will compile and run successfully producing the output shown in Figure 2 .

Note:
Listing 2 - Question 2.

```
/*File Q02.java
**********************************************/
import java.util.ArrayList;
import java.awt.Button;
import java.awt.Label;
import java.awt.Component;
```

```
public class Q02{
/*
Given:
Button and Label are subclasses of the Component
class.
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){
    //Create and populate a list designed to
    // store references to objects of type Button
    // and display its contents.
    ArrayList <Button> listA = new ArrayList<>();
    listA.add(new Button("Button01"));
    listA.add(new Button("Button02"));
    display(listA);
    System.out.println("");//blank line

    //Create and populate a list designed to
    // store references to objects of type Label
    // and display its contents.
    ArrayList <Label> listB = new ArrayList<>();
    listB.add(new Label("Label01"));
    listB.add(new Label("Label02"));
    display(listB);
  }//end main
  //---------------------------------------//

  static void display(
            ArrayList<? extends Component> list){
    for(Component n : list){
      System.out.println(n.getClass());
      if(n instanceof Button){
        System.out.println(
```

```
                            ((Button)n).getLabel());
      }else if(n instanceof Label) {
        System.out.println(((Label)n).getText());
      }//end else
    }//end for loop
  }//end display
}//end class Q02
```

Figure 2 - Question 2.

```
class java.awt.Button
Button01
class java.awt.Button
Button02

class java.awt.Label
Label01
class java.awt.Label
Label02
```

Answer 2

Question 3

True or False: The code shown in Listing 3 will compile and run successfully producing the output shown in Figure 3 .

Note:

Listing 3 - Question 3.

```
/*File Q03.java
**********************************************/
import java.util.ArrayList;
import java.awt.Button;
import java.awt.Label;
import java.awt.Component;

public class Q03{
/*
Given:
Button and Label are subclasses of the Component
class.
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){
    //Create and populate a list designed to
    // store references to objects of type Button
    // and display its contents.
    ArrayList <Button> listA = new ArrayList<>();
    listA.add(new Button("Button01"));
    listA.add(new Button("Button02"));
    display(listA);
    System.out.println("");//blank line

    //Create and populate a list designed to
    // store references to objects of type Label
    // and display its contents.
    ArrayList <Label> listB = new ArrayList<>();
    listB.add(new Label("Label01"));
```

```
    listB.add(new Label("Label02"));
    display(listB);
  }//end main
  //-----------------------------------------//

  static void display(ArrayList<?> list){
    for(Object n : list){
      System.out.println(n.getClass());
      if(n instanceof Button){
        System.out.println(
                          ((Button)n).getLabel());
      }else if(n instanceof Label) {
        System.out.println(((Label)n).getText());
      }//end else
    }//end for loop
  }//end display
}//end class Q03
```

Figure 3 - Question 3.

Figure 3 - Question 3.

```
class java.awt.Button
Button01
class java.awt.Button
Button02

class java.awt.Label
Label01
class java.awt.Label
Label02
```

Answer 3

Question 4

True or False: The code shown in Listing 4 will compile and run successfully producing the output shown in Figure 4 .

Note:
Listing 4 - Question 4.

```
/*File Q04.java
**********************************************/
import java.util.ArrayList;
import java.awt.Button;
import java.awt.Label;
import java.awt.Component;
```

```
public class Q04{
/*
Given:
Button and Label are subclasses of the Component
class.
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){
    //Create and populate a list designed to
    // store references to objects of type Button
    // and display its contents.
    ArrayList <Button> listA = new ArrayList<>();
    listA.add(new Button("Button01"));
    listA.add(new Button("Button02"));
    display(listA);
    System.out.println("");//blank line

    //Create and populate a list designed to
    // store references to objects of type Label
    // and display its contents.
    ArrayList <Label> listB = new ArrayList<>();
    listB.add(new Label("Label01"));
    listB.add(new Label("Label02"));
    display(listB);
  }//end main
  //---------------------------------------//

  static void display(ArrayList<Object> list){
    for(Object n : list){
      System.out.println(n.getClass());
      if(n instanceof Button){
        System.out.println(
                        ((Button)n).getLabel());
```

```
      }else if(n instanceof Label) {
        System.out.println(((Label)n).getText());
      }//end else
    }//end for loop
  }//end display
}//end class Q04
```

Figure 4 - Question 4.

```
class java.awt.Button
Button01
class java.awt.Button
Button02

class java.awt.Label
Label01
class java.awt.Label
Label02
```

Answer 4

Question 5

True or False: The code shown in Listing 5 will compile and run successfully producing the output shown in Figure 5 .

Note:
Listing 5 - Question 5.

```
/*File Q05.java
**********************************************/
import java.util.ArrayList;
import java.awt.Window;
import java.awt.Frame;
import javax.swing.JFrame;

public class Q05{
/*
Given:
Component extends Object
Container extends Component
Window extends Container
Frame extends Window
JFrame extends Frame
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){

    Frame frame0 = new Frame("Frame0");

    ArrayList <Frame> listA = new ArrayList<>();
    listA.add(frame0);
    listA.add(new Frame("Frame1"));
    display(listA);

    ArrayList<Window> listB = new ArrayList<>();
    listB.add(new Window(frame0));
    listB.add(new Window(frame0));
```

```
    display(listB);

    ArrayList<JFrame> listC = new ArrayList<>();
    listC.add(new JFrame("JFrame0"));
    listC.add(new JFrame("JFrame1"));
    display(listC);
  }//end main
  //-----------------------------------------//

  static void display(
                ArrayList<? super JFrame> list){
    for(Object n : list){
      System.out.println(n.getClass());
    }//end for loop
    System.out.println();//blank line
  }//end display
}//end class Q05
```

Figure 5 - Question 5.

```
class java.awt.Frame
class java.awt.Frame

class java.awt.Window
class java.awt.Window

class javax.swing.JFrame
class javax.swing.JFrame
```

Answer 5

Question 6

True or False: The code shown in Listing 6 will compile and run successfully producing the output shown in Figure 6 .

Note:
Listing 6 - Question 6.

```
/*File Q06.java
**********************************************/
import java.util.ArrayList;
import java.awt.Window;
import java.awt.Frame;
import javax.swing.JFrame;

public class Q06{
/*
Given:
Component extends Object
Container extends Component
Window extends Container
Frame extends Window
JFrame extends Frame
The getClass method is defined in the Object
class. When called on a reference to an object,
the method returns the name of the class from
which the object was instantiated.
*/

  public static void main(String[] args){
```

```
    Frame frame0 = new Frame("Frame0");

    ArrayList <Frame> listA = new ArrayList<>();
    listA.add(frame0);
    listA.add(new Frame("Frame1"));
    display(listA);

    ArrayList<Window> listB = new ArrayList<>();
    listB.add(new Window(frame0));
    listB.add(new Window(frame0));
    display(listB);

    ArrayList<JFrame> listC = new ArrayList<>();
    listC.add(new JFrame("JFrame0"));
    listC.add(new JFrame("JFrame1"));
    display(listC);
  }//end main
  //-----------------------------------------//

  static void display(
                 ArrayList<? super Frame> list){
    for(Object n : list){
      System.out.println(n.getClass());
    }//end for loop
    System.out.println();//blank line
  }//end display
}//end class Q06
```

Figure 6 - Question 6.

Figure 6 - Question 6.

```
class java.awt.Frame
class java.awt.Frame

class java.awt.Window
class java.awt.Window

class javax.swing.JFrame
class javax.swing.JFrame
```

Answer 6

Figures

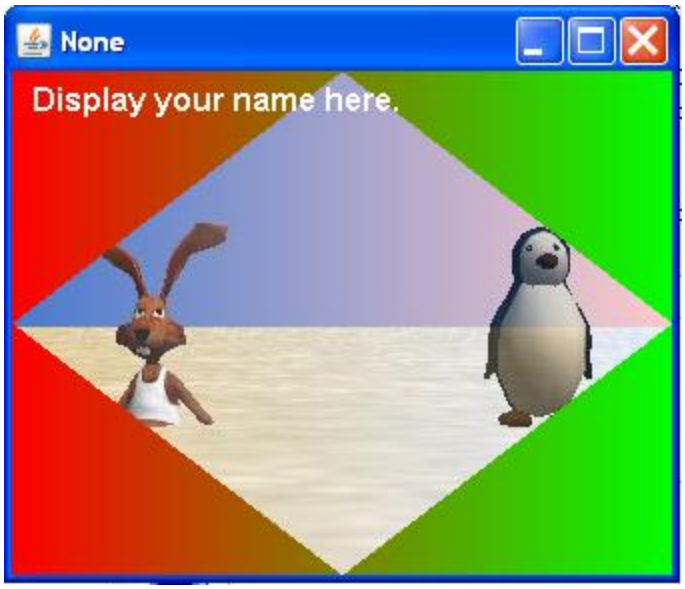
- Figure 1 . Question 1.
- Figure 2 . Question 2.
- Figure 3 . Question 3.
- Figure 4 . Question 4.
- Figure 5 . Question 5.
- Figure 6 . Question 6.
- Figure 7 . Answer 6.
- Figure 8 . Answer 4.
- Figure 9 . Answer 1.

Listings

- Listing 1 . Question 1.
- Listing 2 . Question 2.
- Listing 3 . Question 3.
- Listing 4 . Question 4.
- Listing 5 . Question 5.
- Listing 6 . Question 6.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 6

False. This program attempts to illustrate the use of a lower bounded wildcard to define a generic method that will accept an incoming reference to an object of a generic class where the parameter type for the object is equal to or above the type of the wildcard in the inheritance hierarchy. The lower bounded wildcard type for the display method is Frame. Therefore, the display method will accept references to ArrayList objects for parameter types Frame, Window, Container, Component, or Object, but it will not accept a reference to an ArrayList object for parameter type JFrame. Therefore, the program will not compile, resulting in the error shown in Figure 7 .

Figure 7 - Answer 6.

```
Q06.java:56: error: method display in class
Q06 cannot be applied
to given types;
    display(listC);
    ^
  required: ArrayList<? super Frame>
  found: ArrayList<JFrame>
  reason: actual argument ArrayList<JFrame>
cannot be converted to
  ArrayList<? super Frame> by method
invocation conversion
1 error
```

Back to Question 6

Answer 5

True. This program illustrates the use of a lower bounded wildcard to define a generic method that will accept an incoming reference to an object of a generic class where the parameter type for the object is equal to or above the type of the wildcard in the inheritance hierarchy. The lower bounded wildcard type for the display method is JFrame. Therefore, the display method will accept references to ArrayList objects for parameter types JFrame, Frame, Window, Container, Component, or Object

Back to Question 5

Answer 4

False. This program illustrates the fact that a parameter of type Object is not the same as an unbounded wildcard parameter. The program refuses to compile and produces the errors shown in Figure 8 .

Figure 8 - Answer 4.

Figure 8 - Answer 4.

```
Q04.java:31: error: method display in class
Q04 cannot be applied
to given types;
    display(listA);
    ^
  required: ArrayList<Object>
  found: ArrayList<Button>
  reason: actual argument ArrayList<Button>
cannot be converted to
  ArrayList<Object> by method invocation
conversion

Q04.java:40: error: method display in class
Q04 cannot be applied
to given types;
    display(listB);
    ^
  required: ArrayList<Object>
  found: ArrayList<Label>
  reason: actual argument ArrayList<Label>
cannot be converted to
  ArrayList<Object> by method invocation
conversion
2 errors
```

Back to Question 4

Answer 3

True. This program illustrates two scenarios where an unbounded wildcard is a useful approach:

1. When you are writing a method that can be implemented using functionality provided in the Object class. The getClass method that is called in the display method is defined in the Object class.

2. When the code in a generic method doesn't depend on the type parameter. An if-else statement along with specific casts is used in the display method to isolate the calls to the getText and getLabel methods from the type parameter.

Back to Question 3

Answer 2

True. This program illustrates the use of upper bounded wildcards. The method named display will accept and process a reference to an ArrayList object whose element types are Component or any subclass of Component. Methods that are defined in or inherited into the Component class can be called on the elements with no requirement for a cast. Methods that are not defined in or inherited into the Component class but are defined in the element classes can be called on the elements after casting to the element type.

Back to Question 2

Answer 1

False. This program will not compile. The method named display will only accept and process a reference to an ArrayList object whose element are of type Component. ArrayList objects whose element types are subclasses of Component do not satisfy that requirement. An attempt to compile the program produces the errors shown in Figure 9 .

Figure 9 - Answer 1.

```
Q01.java:34: error: method display in class
Q01
cannot be applied to given types;
    display(listA);
    ^
  required: ArrayList<Component>
  found: ArrayList<Button>
  reason: actual argument ArrayList<Button>
cannot be converted to
  ArrayList<Component> by method invocation
conversion

Q01.java:43: error: method display in class
Q01 cannot be applied
to given types;
    display(listB);
    ^
  required: ArrayList<Component>
  found: ArrayList<Label>
  reason: actual argument ArrayList<Label>
cannot be converted to
  ArrayList<Component> by method invocation
conversion
2 errors
```

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java4270r: Review of Wildcards
- File: Java4270r.htm
- Published: 10/24/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jy0035: Java OOP: Preface to Event Handling
This module serves as the Preface to material for the Event Handling portion of the course, ITSE2317 - Java Programming (Intermediate), which Prof. Baldwin teaches at Austin Community College in Austin, TX.

Revised: Thu May 12 10:35:00 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Additional material

While most of the material required for success in the **Event Handling** portion of this course is contained in this book, other important material is contained in the section titled **Essence of OOP** in the book for the prerequisite course, ITSE 2321 - Object-Oriented Programming (Java) .

In addition, the following material in the **Essence of OOP** section of this book is particularly germane to the **Event Handling** portion of this course:

- Java1636 Java OOP: Member Classes
- Java1638 Java OOP: Local Classes
- Java1640 Java OOP: Anonymous Classes

*(Note in particular that the page titled **Java1640 Java OOP: Anonymous Classes** contains quite a lot of material on event handling in addition to the material on anonymous classes. You will need to understand the material on anonymous classes as well as the material on event handling to succeed in the **Event Handling** portion of this course.)*

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jy0035: Java OOP: Preface to Event Handling
- File: Jy0035.htm
- Published: 10/26/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0073 Java OOP The AWT and Swing, A Preview
This module provides a very brief preview of some of what you can expect to find in subsequent modules regarding the Abstract Windows Toolkit (AWT) and the Swing component set.

Revised: Thu May 12 10:07:20 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Preface
- Discussion
- What's next?
- Miscellaneous

Preface

This module is part of a collection of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

(Editor's note: As you will see when you read this module, the original version was written about eighteen years ago. However, despite numerous improvements that have occurred in Java since then, most of what was true on this topic in 1998 is still true in 2012. A wise man once said, "If it ain't broke, don't fix it."

One of the changes that you will see in subsequent modules is that the name of the package that contains the ***Swing*** *classes has been changed to* ***javax.swing*** *.)*

Discussion

This module provides a very brief preview of some of what you can expect to find in subsequent modules regarding the Abstract Windows Toolkit (**AWT**) and the **Swing** component set.

The user interface of a modern computer program often involves techniques to activate many of the human senses. We use *icons, text boxes, images, sound, boardroom graphics, etc.*

We weren't too concerned with these aspects of programming in the prerequisite course *(ITSE 2321)* because there was a lot that you needed to learn to prepare yourself for understanding material of this sort. That is about to change.

Much of the actual programming that you will do with Java will involve those aspects of the interface that we commonly refer to as the **Graphical User Interface (GUI)** .

As of 5/10/98, there are two primary packages that are used for **GUI** programming under JDK 1.1.6:

1. java. **awt** .*
2. com.sun.java. **swing** .*

There are, of course, numerous other packages that are used in support of these two packages.

The **AWT** material was made available to Java programmers early in the life of Java. This was the original material that was used to create graphical user interfaces. Major improvements to the **AWT** were introduced with the release of JDK 1.1.

The **Swing** components became available in released form for use with JDK 1.1 around the beginning of 1998. These components added significantly to the ability of the programmer to create GUIs, both in terms of functionality and cosmetics.

The capability and cosmetics of the **AWT** were very limited but **Swing** made GUI programming in Java competitive in the real world. A Java

programmer no longer need apologize for the quality of the GUIs that she can create.

We expect that these two packages may become more integrated (causing changes in your **import** directives) with the release of JDK 1.2, *(probably sometime in 1998)* but hopefully the concepts involved won't be greatly different.

As of 3/5/97, there were more than fifty classes defined in package **java.awt** . We will discuss some of the more important **AWT** classes in subsequent modules.

As of 5/10/98, the *com.sun.java.swing* package contains more than 75 classes and about 20 interfaces. You might expect, therefore, that learning to use this material effectively won't be a trivial task.

It is very important to understand that **Swing** is an extension of, and not a replacement for the **AWT** . While it is true that there is some overlap *(for example a **Swing JButton** component might be viewed as an improved functional replacement for an **AWT Button** component, and once you begin using **Swing** buttons you may choose to never again use an **AWT** button)* , the basic functionality of **Swing** is built upon the functionality of the **AWT** .

Therefore, as students, we cannot simply skip over an understanding of the **AWT** and move on to **Swing** . The **AWT** is the foundation for **Swing** .

We must first understand the **AWT** and then understand how **Swing** extends and improves on the **AWT** . I will attempt to integrate an understanding of both the **AWT** and **Swing** in the remaining modules in this collection.

We will begin by introducing you to a few simple components of each type and use these components to teach you about such topics as event-driven programming, layout, graphics, etc. Then, time permitting, we will dig a little deeper into the more complex aspects of both the **AWT** and **Swing** components and other features.

What I won't do is show you a lot of pictures of various **AWT** and **Swing** components as is the case with many books and other tutorials *(although*

such pictures can be important for an appreciation of GUI programming) . (Have you noticed how many Java books use copies of the JavaSoft documentation as filler material to make the book appear to contain more information than it actually contains? At least half of many of the books currently in print is nothing more than a reproduction of the documentation that you can download for free from JavaSoft. Oh well, enough of that!)

If you want to see some pictures of **AWT** and **Swing** components *(which would be only natural)* , you can create them yourself on your own computer screen.

For examples of the **AWT** components, simply look in the folders in the software that you downloaded from JavaSoft. When you install JDK 1.1.6, a folder named "demo" will be created that contains about two-dozen sample programs. Many of these sample programs have graphical user interfaces that make use of the **AWT** . Just run the programs to see examples of the use of the **AWT** .

When you download and install **Swing** 1.0.1, a folder named "examples" will be created. This folder contains about nine folders, each of which contains a demonstration application or applet that makes use of **Swing** . You can run these programs to see the examples on your computer screen.

A particularly interesting demonstration application is the one named **SwingSet** . One of the new components in **Swing** is a tabbed pane that looks much like a common cardboard file folder with a labeled tab on the top, bottom, left, or right. The **AWT** doesn't contain such a component.

This demonstration starts with about twenty such tabbed panes on the screen, each one of which demonstrates one aspect of the use of **Swing** . By clicking on each of the labeled tabs, you can select and exercise one aspect of **Swing** . In addition, there are five menus that contain selections, some of which impact the behavior of some aspect of the demonstration.

While you are there, pay attention to the fact that virtually all of the **Swing** components are also containers, so it is possible to cause other items *(such as images)* to be contained in components such as buttons and menus.

Take a look at the pane labeled *RadioButtons* and see how two different images of JavaSoft's little creature named Duke can be made to function as a radio button. In this case, the selected Duke is waving while the unselected Dukes aren't waving.

Duke shows up again under *ToggleButtons* where the button which has been toggled has Duke animated in a child's swing.

The *Checkboxes* pane uses light bulbs that either are or are not illuminated to illustrate selection of **Checkbox** items.

The examples on the *Slider* pane are truly impressive *(the **AWT** doesn't have a slider component, although it is possible to use a **ScrollBar** as a crude slider)* .

Take a look at the *ListBox* pane to see another example of using images inside of a component.

The *DebugGraphics* pane demonstrates how to run your program in slow motion so that you can see how the components are assembled for debugging purposes. Note that a **Slider** is used to control the speed of assembly of the components.

And of course, every where you turn in this demo, you will see *tool tips* that are not a part of the **AWT** . For a little comic relief, take a look at the *ToolTips* pane.

Don't forget to pull down the *Options* menu and select the "look and feel" of the different panes as you view them.

Actually, words are inadequate to describe what you are going to find when you install and run the **SwingSet** demonstration. To use a corny phrase made famous by an old TV commercial *(which many of you are probably too young to remember)* , "Try it, you'll like it."

What's next?

The next module in the collection will take a first look at *callbacks* .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: The AWT and Swing, A Preview
- File: Java0073.htm
- Published: 1998

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0077 Java OOP Callbacks - I
Learn the fundamentals of callbacks using interfaces in Java.

Revised: Thu May 12 11:00:55 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
    - Unicast sample program
    - Multicast sample program
- Run the program
- Summary
- What's next?
- Miscellaneous

Preface

This module is one in a series of three modules designed to teach you about callbacks in Object-Oriented Programming (OOP) using Java. The other two modules are titled Callbacks - II and Callbacks - III.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Documentation for the removeElement method.
- Figure 2 . Output from Callback02.

Listings

- Listing 1 . Define the CallBack interface.
- Listing 2 . Define the Teacher class.
- Listing 3 . Define the Student class.
- Listing 4 . A controlling class named Callback01.
- Listing 5 . Complete listing of program named Callback01.
- Listing 6 . Define the CallBack interface.
- Listing 7 . Define the Teacher class.
- Listing 8 . Define the method named register.
- Listing 9 . The unRegister method.
- Listing 10 . Define the callTheRoll method.
- Listing 11 . Define the class named Dog.
- Listing 12 . Define the class named Callback02.
- Listing 13 . Complete listing of program named Callback02.

Preview

Many processes in the standard Java API make use of a mechanism that might be referred to as a *callback* mechanism. Basically, this is a mechanism where a method in one object asks a method in another object to *"call me back"* or *"notify me"* when an interesting event happens.

An interesting event

For example, an interesting event might be that the price of a specified stock goes above its previous high value, or the toaster finishes toasting the bread.

Multicasting

In fact, many different objects may ask one object to notify them when the interesting event happens. This is sometimes referred to as *multicasting* . *(The one-to-one case is often referred to as unicasting.)*

Going further, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification.

Many examples

For example, we see different forms of callback activity in conjunction with

- the *Delegation Event Model* used with GUIs in JDK 1.1,
- the *Observer/Observable* concept used in the *Model-View-Controller* paradigm,
- the concept of *Bound Properties* and *Constrained Properties* in *Java Beans* , etc.

You can find examples of all of these in the pages of my online tutorial lessons.

Callback implementation

Callback capabilities are often implemented in other languages by passing a function pointer to another function. The receiving function uses the passed function pointer to call another function when an interesting event happens. However, Java doesn't support function pointers. In this module, we will learn how to implement the callback mechanism using **interfaces** instead.

From the simple to the more complex

As usual, our approach will be to learn the material by reviewing programs that progress from very simple to more complex. As mentioned earlier, this topic consumes all of this module and two additional lessons on my website as well.

Meaningful scenarios

It is usually easier to understand abstract concepts if they are explained in terms of meaningful scenarios. In this case, our scenario will consist of a *teacher* and some *students* . In the beginning there will only be one student. Ultimately there will be many students and there will also be some animals in the classroom as well.

Registration

The students *(and the animals)* register themselves on the teachers roll book to be notified of interesting events. Initially the interesting event will simply be the teacher taking the roll. Ultimately the interesting event will be notification that it is *either* time for recess, *or* it is time for lunch.

Unicast and multicast scenarios

Initially, only one student receives notification of the one type of event. Ultimately, all of the students and all of the animals receive notification of both types of event *(recess or lunch)* but some of those who are notified choose to ignore the notification.

We will refer to the case where only one student is on the list as the *unicast* program. We will refer to the case where many students *(and possibly animals as well)* are on the list as the *multicast* program. This terminology was selected because it matches the terminology used in the JDK 1.1 documentation for the Delegation Event Model.

Without further discussion, let's look at some code.

Discussion and sample code

Unicast sample program

The purpose of this program is to develop a *callback* capability using *Interfaces* . This version of the program is designed to emphasize the structure of the process. Therefore an effort was made to avoid the requirement for any extra code so it doesn't do anything fancy.

A CallBack interface

This program defines a **CallBack** interface *(interface named **CallBack***) that can be used to establish a new type of object reference, and also to declare the interface to a method named **callBack** that will be contained in all objects of classes that implement the interface. This method will then be used to notify those objects whenever something interesting happens.

A Teacher class

The program defines a **Teacher** class that has the ability to

- create and maintain a list of *(only)* one object of the interface type *(multiple objects come later)* , and
- to notify that object that something interesting has happened by calling its **callBack** method.

(As mentioned earlier, the size of the list was constrained to only one object in order to emphasize callback structure and avoid getting bogged down in list processing. A subsequent version will implement list processing.)

A Student class

The program defines a class named **Student** that *implements* the **CallBack** interface. Objects of the **Student** class can be *registered* on the list maintained by an object of the **Teacher** class, and can be *notified* by the object of the **Teacher** class whenever something interesting happens. Notification takes the form of calling the **callBack** method on the object.

The method named callBack

The body of the **callBack** method can be designed to do anything, but in this case, to keep things simple, it just announces that it has been called.

The controlling class

Finally, the program defines a controlling class named **Callback01** that ties all the pieces together and exercises them.

The program was originally tested using JDK 1.1.3 under Win95 and more recently tested using JDK 1.7 under Windows Vista.

The output from the program is shown in the complete program listing in a later section.

Interesting unicast code fragments

Listing 1 defines an interface named **CallBack** that creates a new type and declares a generic method named **callBack** that can be used to execute a callback on any object that is instantiated from a class that implements the interface.

Listing 1 . Define the CallBack interface.

```
interface CallBack{
  public void callBack();
}//end interface CallBack
```

A class that can register and notify objects of type CallBack

Next we need a class whose objects can maintain a list of references to objects of type **CallBack** *(objects whose class implements the **CallBack***

interface) .

We refer to the process of putting an object on the list is *registering* the object.

This class also needs to have the ability to notify all the objects on that list when something interesting happens. We will name this class **Teacher** in keeping with the scenario described earlier.

As mentioned earlier, to keep things simple, and emphasize the callback structure without getting bogged down in list processing, we will begin with a limitation of one object for the length of the list.

The unicast class named Teacher

The unicast **Teacher** class consists of one instance variable of type **CallBack** *(the interface type)* and two instance methods.

One of the methods named **register** places an object on the list. The other method named **callTheRoll** calls the **callBack** method on the object that is on the list.

Note that the object on the list is guaranteed to have a method named **callBack** because it implements the **CallBack** interface. Otherwise, it couldn't get on the list in the first place. This is because the **register** method requires the incoming object's reference to be of type **CallBack** .

The Teacher class is defined in Listing 2 .

Listing 2 . Define the Teacher class.

Listing 2 . Define the Teacher class.

```
class Teacher{
  CallBack obj; //list of objects of type
CallBack
  //-----------------------------------------
----------//

  //Method to add objects to the list.
  void register(CallBack obj){
    this.obj = obj;
  }//end register()
  //-----------------------------------------
----------//

  //Method to notify all objects on the list
  void callTheRoll(){
    obj.callBack();
  }//end callTheRoll()
  //-----------------------------------------
----------//
}//end class Teacher
```

A class that implements the CallBack interface

Next, we need a class that *implements* the **CallBack** interface. Objects of this class can be registered on the list maintained by an object of the **Teacher** class, and will be *notified* whenever that object calls the **callBack** method on the registered objects on the list. In keeping with the scenario described earlier, we will name this class **Student** .

By claiming to implement the **CallBack** interface, this class is required to provide a concrete definition for the method named **callBack** that is declared in the interface. Otherwise, the program won't compile. In this

case, that definition is rather simple. The **callBack** method simply announces that it has been called.

The callback mechanism

As we saw above, an object of the **Teacher** class will call the **callBack** method on all objects on its list when the interesting event occurs. It is important to note that the callback mechanism is to call this method.

The **Student** class is defined in Listing 3 .

Listing 3 . Define the Student class.

```
class Student implements CallBack{
  String name;
  //-----------------------------------------
----------//

  Student(String name){//constructor
    this.name = name;  //save the name to
identify the obj
  }//end constructor
  //-----------------------------------------
----------//

  public void callBack(){
    System.out.println(name + " here");
  }//end callBack()
}//end class Student
```

A controlling class named Callback01

Finally, we need a controlling class to tie all the pieces together and to exercise them. The **main** method in this class

- instantiates an object of the **Teacher** class named **missJones** ,
- instantiates an anonymous **Student** object named " **Joe** ",
- registers the object on the list maintained by **missJones** , and
- calls the **callTheRoll** method on **missJones** to cause the objects on the list to be notified (to cause their **callBack** methods to be called).

This is not too complicated once you break the process into its component parts.

The class named Callback01 is defined in Listing 4 .

Listing 4 . A controlling class named Callback01.

```
class Callback01{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();
    //Instantiate and register a Student
object with the
    // Teacher object
    missJones.register(new Student("Joe"));
    //Cause the Teacher object to do a
callBack on the
    // Student object.
    missJones.callTheRoll();
  }//end main()
}//end class Callback01
```

There you have it. This simple program contains the sum and substance of one approach to callbacks in Java.

It is critical to note that the objects registered on the list are of the interface type **CallBack** . This guarantees that there cannot be an object on the list that does not have an instance method named **callBack** .

Unicast Program Listing

A complete listing of the program is provided in Listing 5 so that you can view the code fragments in context.

Note:
Listing 5 . Complete listing of program named Callback01.

```
/*File Callback01.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a
callback
capability using Interfaces.  This version of the
program is designed to emphasize the structure of
the process, and therefore an effort was made to
avoid the requirement for any extra code to do
anything fancy.

Tested using JDK 1.1.3 under Win95.

The output from the program is:

Joe here.
**********************************************
********/
//First we define an interface that will create a
new type
// and declare a generic method that can be used
to
// callback any object that is of a class that
```

```
implements
// the interface.
interface CallBack{
  public void callBack();
}//end interface CallBack
//===================================================
=======//

//Next we need a class whose objects can maintain
a
// registered list of objects of type CallBack
(whose
// class implements the CallBack interface) and
can
// notify all the objects on that list when
something
// interesting happens.

//To keep things simple, and emphasize the
structure of
// what we are doing, we will begin with a
limitation
// of one object on the length of the list.

class Teacher{
  CallBack obj; //list of objects of type CallBack
  //-----------------------------------------------
-------//

  //Method to add objects to the list.
  void register(CallBack obj){
    this.obj = obj;
  }//end register()
  //-----------------------------------------------
-------//

  //Method to notify all objects on the list that
```

```
  // something interesting has happened.
  void callTheRoll(){
    //Call the callBack() method on the object.
The
    // object is guaranteed to have such a method
because
    // it is of a class that implements the
CallBack
    // interface.
    obj.callBack();
  }//end callTheRoll()
  //-----------------------------------------------
-------//
}//end class Teacher
//=================================================
=======//

//Class that implements the CallBack interface.
Objects
// of this class can be registered on the list
maintained
// by an object of the Teacher class, and will be
notified
// whenever that object calls the callBack method
on the
// registered objects on the list.
class Student implements CallBack{
  String name;
  //-----------------------------------------------
-------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------------
-------//
```

```
  //An object of the Teacher class will call this
method
  // as the callback mechanism to notify an object
of this
  // class that something interesting has
happened.
  public void callBack(){
    System.out.println(name + " here");
  }//end overridden callBack()
}//end class Student
//===================================================
=======//

//Controlling class that ties all the pieces
together and
// exercises them.
class Callback01{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();
    //Instantiate and register a Student object
with the
    // Teacher object
    missJones.register(new Student("Joe"));
    //Cause the Teacher object to do a callBack on
the
    // Student object.
    missJones.callTheRoll();
  }//end main()
}//end class Callback01
//===================================================
=======/
```

Multicast sample program

The multicast version of this program does not modify the basic callback mechanism developed in the previous program. It simply enhances that mechanism to make it possible to maintain *a list of objects* registered for callback and to notify all the objects on that list when an interesting event happens.

In case you started reading at this point, this is an enhanced version of the program named **Callback01** . You should familiarize yourself with that program before trying to understand this program.

A list of registered objects

This program has the capability to create and maintain a list of objects that register for callback whereas the program named **Callback01** could only remember a single object for callback.

Multiple classes implement CallBack interface

In addition, this program defines two different classes that implement the **CallBack** interface. Mixed objects of those two types are maintained on the list and notified at callback time. This is a subtle but very important point. It is not necessary that all the objects that are registered on a callback list be of the same class type, only that they all be of a class that implements the **CallBack** interface.

The CallBack interface

As before, this program defines a **CallBack** interface that establishes a new type of object, and also declares the interface to a method named **callBack** that is contained in all objects of classes that implement the interface. Because the **callBack** method is guaranteed to be contained in all of the objects on the list, it can be used to notify those objects whenever something interesting happens.

The Teacher class

The program defines a **Teacher** class that creates and maintains a list of objects of the **CallBack** interface type, and notifies those objects that something interesting has happened by calling the **callBack** method on each of the objects on the list.

The size of the list is limited only to the largest **Vector** object that can be accommodated by the system. *(See the Java documentation or my online tutorials for information about the **Vector** class.)*

The Student and Dog classes

The program defines a class named **Student** that implements the **CallBack** interface. The program also defines a class named **Dog** that implements the **CallBack** interface as well. *(Back in the description of the scenario, I promised you that **missJones** was going to have to deal with animals in the classroom. I'm glad I don't have that problem.)*

Registration and notification of Student and Dog objects

Objects of the **Student** and **Dog** classes can be *registered* on the list *(of **CallBack** objects)* maintained by an object of the **Teacher** class *(because they both implement the **CallBack** interface)* , and can be *notified* by the object of the **Teacher** class whenever something interesting happens.

Addition and removal from the list

Note that objects can be added to the list and then removed from the list. One object is first added and later removed for demonstration purposes.

The callback mechanism

As before, notification takes the form of calling the **callBack** method on each of the objects on the list.

Behavior of the callBack methods

The behavior of the **callBack** methods in the classes that implement the interface can be designed to do anything. In this case, to keep things simple,

they just announce that they have been called. However, they make the announcement in slightly different ways.

Text display statements

This program contains display statements in the registration and notification methods for demonstration purposes only, and to allow us to track what is happening as the program runs.

The controlling class

Finally, the program defines a controlling class named **Callback02** that ties all the pieces together and exercises them.

The program was originally tested using JDK 1.1.3 under Win95 and more recently tested using JDK 1.7 under Windows Vista.

The output from the program is shown following a discussion of the controlling class at the end of the next section.

Interesting multicast code fragments

Listing 6 defines an interface that creates a new type and declares a generic method that can be used to call back any object that is of a class that implements the interface. There is nothing new here.

Listing 6 . Define the CallBack interface.

```
interface CallBack{
  public void callBack();
}//end interface CallBack
```

A class that can register and notify objects of type CallBack

Next we need a class whose objects can maintain a registered list of objects of type **CallBack** *(objects whose class implements the **CallBack** interface)* and can *notify* all the objects on that list when something interesting happens. As before, we name this class **Teacher** .

The Teacher class

The **Teacher** class has grown to the point that we will break it into parts and discuss them separately.

There is quite a bit here that is new, due simply to the requirement for list processing. There is nothing new about the basic callback mechanism.

An object of type Vector

We start out by replacing the single instance variable of type **CallBack** by a reference to an object of type **Vector** . We will maintain our list in an object of type **Vector** .

Recall that a **Vector** object can only work with references to objects of type **Object** , so this will entail some down casting later.

*(Editor's note: Sometime around JDK 1.5, a concept known as generics was released into Java, which eliminated the restriction to objects of type **Object** mentioned in the previous paragraph. However, this code has not been updated to take advantage of that capability.)*

The constructor for our new **Teacher** class, which is shown in Listing 7 , instantiates the **Vector** object.

Listing 7 . Define the Teacher class.

Listing 7 . Define the Teacher class.

```
class Teacher{
  Vector objList; //list of objects of type
CallBack
  //---------------------------------------
----//

  Teacher(){//constructor
    objList = new Vector();
  }//end constructor
```

The method named register

Next we need a method to add objects to the list. We will synchronize it to protect against the possibility of two or more objects on different threads trying to register at the same time.

Note that the references to the objects are received as type **CallBack** , which is the interface type, and stored as type **Object** , because the **Vector** class only accommodates references to objects of type **Object** . *(See the earlier editor's note.)* Again, this will lead to some down casting requirements later.

Listing 8 . Define the method named register.

Listing 8 . Define the method named register.

```
synchronized void register(CallBack obj){
  this.objList.addElement(obj);
  System.out.println(obj + " added");
}//end register()
```

The unRegister method

To be general, we also need a method to remove objects from the list. Removal of an object from the list is a little more complicated than adding an object to the list due to the possibility of having two or more identical objects on the list. *(We could, and possibly should, guard against that possibility when constructing the list.)*

Figure 1 contains a partial excerpt from the JDK 1.1.3 documentation, which describes the **removeElement** method of the **Vector** class that we are using to accomplish this *(three different methods are available to remove objects from a **Vector***).

Figure 1 . Documentation for the removeElement method.

Figure 1 . Documentation for the removeElement method.

```
public final synchronized boolean
removeElement(Object obj)

This method removes the first occurrence of
the argument from this
vector. Indices beyond that point are adjusted
appropriately

Parameters: obj - the component to be removed.

Returns: true if the argument was a component
of this vector;
         false otherwise.
```

Registered object removal code

Given that explanation, the code for removal of an object from the list is straightforward. The **unRegister** method is shown in Listing 9 .

Listing 9 . The unRegister method.

Listing 9 . The unRegister method.

```
  synchronized void unRegister(CallBack obj){
    if(this.objList.removeElement(obj))
      System.out.println(obj + " removed");
    else System.out.println(obj + " not in the
list");
  }//end register()
```

The callTheRoll method

Now we need a method to notify all of the objects on the list that something interesting has happened. We will name this method **callTheRoll** to adhere to our classroom scenario.

Maintain the integrity of the callback list

One of the potential problems with this type of callback mechanism is that when the callback method is called on an object, that method might take a while to finish.

(As an aside, when writing callback methods, if they do anything significant in terms of time, the code in the method should probably spawn another thread to do the actual work and return as quickly as possible.)

This leads to the possibility that additional objects might attempt to register during that time interval. To protect against this, we make a copy of the state of the list object as it existed at the point in time that the decision was made to do the callbacks, and then perform the callbacks using that copy. That way, the original list is free to be updated as needed during this interval.

So, we start out by creating a clone of the list. We also *synchronize* this process to prevent the list from being modified while we are creating the clone.

Following this, we use a **for** loop to access all the objects on the list, and call the **callBack** method on those objects. *(Actually, the list contains references to objects, and not the actual objects, so we are calling the method on the references.)*

As promised earlier, we have to downcast from **Object** to **CallBack** to gain access to the **callBack** method in the objects.

Listing 10 . Define the callTheRoll method.

```
  void callTheRoll(){
    Vector tempList;//save a temporary copy of
list here

    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    for(int cnt = 0; cnt < tempList.size();
cnt++){

((CallBack)tempList.elementAt(cnt)).callBack();
    }//end for loop
  }//end callTheRoll()
```

End of the class named Teacher

That ends the discussion of the class named **Teacher** and brings us to the class named **Student** that implements the **CallBack** interface. This class

hasn't changed. As indicated earlier, this version of the program also has a class named **Dog** that implements the interface. These two classes are essentially the same.

Define the class named Dog

Because of their similarity, and because they are essentially the same as in the previous program, I will simply show the class named **Dog** with no further discussion.

Listing 11 . Define the class named Dog.

Listing 11 . Define the class named Dog.

```
class Dog implements CallBack{
  String name; //store name here for later ID
  //----------------------------------------
----------//

  Dog(String name){//constructor
    this.name = name; //save the name to
identify the obj
  }//end constructor
  //----------------------------------------
----------//

  //An object of the Teacher class will call
this method
  // as the callback mechanism to notify an
object of this
  // class that something interesting has
happened.

  public void callBack(){//announce callBack

    System.out.println("Woof, Woof " + name);
  }//end overridden callBack()
}//end class Dog
```

The controlling class

That brings us to the controlling class named **Callback02** that ties all the pieces together and exercises them. This class is shown in Listing 12 .

Note:

Listing 12 . Define the class named Callback02.

```
class Callback02{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

 //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");

    //Instantiate some Dog objects.
    Dog spot = new Dog("Spot");
    Dog fido = new Dog("Fido");
    Dog brownie = new Dog("Brownie");

    //Register some Student and Dog objects with
the
    // Teacher object.
    System.out.println("Register Tom");
    missJones.register(tom);
    System.out.println("Register Spot");
    missJones.register(spot);
    System.out.println("Register Sue");
    missJones.register(sue);
    System.out.println("Register Fido");
    missJones.register(fido);
    System.out.println("Register Peg");
    missJones.register(peg);
    System.out.println("Register Bob");
    missJones.register(bob);
    System.out.println("Register Brownie");
    missJones.register(brownie);
```

```
    //Remove a Student object from the list.
    System.out.println("Remove Peg");
    missJones.unRegister(peg);

    //Try to remove an object that is not on the
list.
    System.out.println("Try to remove Joe");
    missJones.unRegister(joe);

    System.out.println();//blank line

    //Cause the Teacher object to do a callBack on
all
    // the objects on the list.
    missJones.callTheRoll();
  }//end main()
}//end class Callback02
```

Differences relative to Callback01

This program differs from the previous program primarily in terms of the volume of **Student** and **Dog** objects to be instantiated and registered on the **Teacher** object. There are also a lot of display statements to help us keep track of what is going on.

The ability to remove objects from the list is also illustrated.

Call the roll

Finally, the callback to the objects on the list is executed in Listing 12 by calling the **callTheRoll** method on the **Teacher** object named **missJones** . The output from running this program is shown later.

Mixed object types

A subtle, but extremely important point is illustrated here. **Student** and **Dog** are different classes. Objects of both of those classes are registered on the single object of the **Teacher** class. The **Teacher** object doesn't care that they are different, so long as they are all instantiated from classes that implement the **CallBack** interface. The **register** method will only accept object references of type **CallBack** .

Program output

The output from running this program is shown in Figure 2 . You can see the identification of each individual object as it is added to, or removed from the list.

Figure 2 . Output from Callback02.

Figure 2 . Output from Callback02.

```
Register Tom
Student@1cc73e added
Register Spot
Dog@1cc74e added
Register Sue
Student@1cc741 added
Register Fido
Dog@1cc751 added
Register Peg
Student@1cc744 added
Register Bob
Student@1cc747 added
Register Brownie
Dog@1cc754 added
Remove Peg
Student@1cc744 removed
Try to remove Joe
Student@1cc74a not in the list

Tom here
Woof, Woof Spot
Sue here
Woof, Woof Fido
Bob here
Woof, Woof Brownie
```

Note that the attempt to remove Joe from the list was not successful because he was never registered in the first place.

Finally, you see the output produced by calling **callTheRoll** which in turn calls the **callBack** method on each of the objects on the list.

Note that Peg didn't appear in the roll call because she was first added and then removed from the list before the roll call was taken.

The sum and substance

So there you have it, the sum and substance of multicast callbacks in Java. Obviously improvements could be made. You can see a couple of them in the remaining two tutorial lessons on callbacks that are published on my website.

Multicast Program Listing

A complete listing of the multicast program named **Callback02** is provided in Listing 13 .

Note:
Listing 13 . Complete listing of program named Callback02.

```
/*File Callback02.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a
callback
capability using Interfaces.

This is an enhanced version of the program named
Callback01. You should familiarize yourself with
the earlier program before getting into this
program.

This version has the added capability to create
and
maintain a list of objects that register for
callback
whereas the program named Callback01 could only
remember
a single object for callback.
```

```
Tested using JDK 1.1.3 under Win95.

The output from the program was:

Register Tom
Student@1cc73e added
Register Spot
Dog@1cc74e added
Register Sue
Student@1cc741 added
Register Fido
Dog@1cc751 added
Register Peg
Student@1cc744 added
Register Bob
Student@1cc747 added
Register Brownie
Dog@1cc754 added
Remove Peg
Student@1cc744 removed
Try to remove Joe
Student@1cc74a not in the list

Tom here
Woof, Woof Spot
Sue here
Woof, Woof Fido
Bob here
Woof, Woof Brownie

Note that Peg didn't appear in the callBack list
because
she was first added to, and later removed from the
list.
**********************************************
********/
import java.util.*;
```

```
//First we define an interface that will create a
new type
// and declare a generic method that can be used
to
// callback any object that is of a class that
implements
// the interface.
interface CallBack{
  public void callBack();
}//end interface CallBack
//===================================================
=======//

//Next we need a class whose objects can maintain
a
// registered list of objects of type CallBack
(whose
// class implements the CallBack interface) and
can
// notify all the objects on that list when
something
// interesting happens.

class Teacher{
  Vector objList; //list of objects of type
CallBack
  //-----------------------------------------------
-------//

  Teacher(){//constructor
    //Instantiate a Vector object to contain the
list
    // of registered objects.
    objList = new Vector();
  }//end constructor
  //-----------------------------------------------
```

```
-------//

  //Method to add objects to the list.
Synchronize to
  // protect against two or more objects on
different
  // threads trying to register at the same time.
Note
  // that the objects are received as type
CallBack which
  // is the interface type, and stored as type
Object,
  // because the Vector class only accommodates
objects of
  // type Object.
  synchronized void register(CallBack obj){
    this.objList.addElement(obj);
    System.out.println(obj + " added");
  }//end register()
  //-----------------------------------------
-------//

  //Method to remove objects from the list.
  synchronized void unRegister(CallBack obj){
    if(this.objList.removeElement(obj))
      //true when successfully found and removed
      System.out.println(obj + " removed");
    else//false on failure to find and remove
      System.out.println(obj + " not in the
list");
  }//end register()
  //-----------------------------------------
-------//

  //Method to notify all objects on the list that
  // something interesting has happened.
  void callTheRoll(){
```

```
    Vector tempList;//save a temporary copy of
list here

    //Make a copy of the list to avoid the
possibility of
    // the list changing while objects are being
notified.
    // Synchronize to protect against list
changing while
    // making the copy.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Call the callBack() method on each object on
    // the list.  The object are guaranteed to
have such
    // a method, even if they are of different
types,
    // because they are all of a class that
implements
    // the CallBack interface. If not, they could
not
    // have been registered on the list in the
first
    // place.  Note the requirement to downcast to
    // type CallBack.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{

((CallBack)tempList.elementAt(cnt)).callBack();
    }//end for loop
  }//end callTheRoll()
  //-----------------------------------------------
-------//
}//end class Teacher
//===============================================
```

```
=======//

//Class that implements the CallBack interface.
Objects
// of this class can be registered on the list
maintained
// by an object of the Teacher class, and will be
notified
// whenever that object calls the callBack method
on the
// registered objects on the list.  This program
will not
// compile if this class fails to implement the
CallBack
// interface

class Student implements CallBack{
  String name; //store the object name here for
later ID
  //-----------------------------------------------
-------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------------
-------//

  //An object of the Teacher class will call this
method
  // as the callback mechanism to notify an object
of this
  // class that something interesting has
happened.

  public void callBack(){//announce callBack
```

```
    System.out.println(name + " here");
  }//end overridden callBack()
}//end class Student
//===========================================
=======//

//Another Class that implements the CallBack
interface.
// Objects of this class can also be registered on
the list
// maintained by an object of the Teacher class,
and will
// also be notified whenever that object calls the
// callBack() method on the registered objects on
the
// list. This program will not compile if this
class
// fails to implement the CallBack interface.

class Dog implements CallBack{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------
-------//

  //An object of the Teacher class will call this
method
  // as the callback mechanism to notify an object
of this
  // class that something interesting has
happened.
```

```
  public void callBack(){//announce callBack
    System.out.println("Woof, Woof " + name);
  }//end overridden callBack()
}//end class Dog
//===================================================
=======//

//Controlling class that ties all the pieces
together and
// exercises them.
class Callback02{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

    //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");

    //Instantiate some Dog objects.
    Dog spot = new Dog("Spot");
    Dog fido = new Dog("Fido");
    Dog brownie = new Dog("Brownie");

    //Register some Student and Dog objects with
the
    // Teacher object.
    System.out.println("Register Tom");
    missJones.register(tom);
    System.out.println("Register Spot");
    missJones.register(spot);
    System.out.println("Register Sue");
    missJones.register(sue);
```

```
    System.out.println("Register Fido");
    missJones.register(fido);
    System.out.println("Register Peg");
    missJones.register(peg);
    System.out.println("Register Bob");
    missJones.register(bob);
    System.out.println("Register Brownie");
    missJones.register(brownie);

    //Remove a Student object from the list.
    System.out.println("Remove Peg");
    missJones.unRegister(peg);

    //Try to remove an object that is not on the
list.
    System.out.println("Try to remove Joe");
    missJones.unRegister(joe);

    System.out.println();//blank line

    //Cause the Teacher object to do a callBack on
all
    // the objects on the list.
    missJones.callTheRoll();
  }//end main()
}//end class Callback02
//===================================================
=======//
```

Run the program

I encourage you to copy the code from Listing 5 and Listing 13 . Compile the code and execute it. Experiment with the code, making changes, and

observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned the fundamentals of callbacks using interfaces in Java.

What's next?

In the next module, you will learn about something that goes by the name Delegation Event Model along with a few other names as well.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Callbacks - I
- File: Java0077.htm
- Published: 1998

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0078 Java OOP Callbacks - II
A previous lesson introduced you to the basic Java callback mechanism using interfaces, and walked you through the development of a set of classes that implement a simple multicast form of callback. In that lesson, the definition of the CallBack interface was limited to a single method declaration. In a real program involving callbacks, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification. This lesson will enhance our previous program to accommodate this possibility.

Revised: Tue Jul 05 19:16:25 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- Sample program
    - Interesting code fragments
- Summary
- Complete program listing
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

This is the second in a series of three consecutive lessons on *Callbacks* in Java. The three lessons are named Callbacks - I, Callbacks - II, and Callbacks - III. Students in Prof. Baldwin's **ITSE 2317 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Output from the program named Callback03.

Listings

- Listing 1 . The CallBack interface.
- Listing 2 . The callRecess method in the Teacher class.
- Listing 3 . The Dog class.
- Listing 4 . The controlling class named Callback03.
- Listing 5 . The program named Callback03.

Introduction

Many processes in the standard Java API make use of a mechanism that in other programming environments might be referred to as a *callback* mechanism. Basically, this is a mechanism where a method in one object asks a method in another object to *"call me back"* or *"notify me"* when an interesting event happens. For example, an interesting event might be that the price of a specified stock goes above its previous high value, or the toaster finishes toasting the bread.

A previous lesson introduced you to the basic Java callback mechanism using interfaces and walked you through the development of a set of classes that implement a simple multicast form of callback. In that lesson, the definition of the **CallBack** interface was limited to a single method declaration. In a real program involving callbacks, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification.

This lesson will enhance our previous program to accommodate this possibility. As mentioned in the earlier lesson, it is usually easier to understand abstract concepts if they are explained in terms of a meaningful scenario. For that reason, we have conjured up a scenario in which to develop and explain our callback programs.

Our scenario consists of a teacher and some students. In the beginning there was only one student. Then we expanded the scenario to include many students and some animals in the classroom as well. The students *(and the animals)* register themselves on the teachers roll book to be notified of interesting events. Initially the interesting event was simply the teacher taking the roll. In this lesson, we expand that scenario to include notification that it is either time for recess, or it is time for lunch.

Initially, only one student received notification of one type of event. In this lesson, all of the students and all of the animals receive notification of both types of event *(recess or lunch)* but some of those who are notified choose to ignore the notification. Without further discussion, let's look at some code.

Sample program

In case you just started reading at this point, this program named **Callback03** is an enhanced version of the program named **Callback02** that you learned about in an earlier lesson. You should familiarize yourself with the earlier program before trying to understand this program.

The earlier version of the program defined two different classes that implemented the **CallBack** interface. In order to give us more to work with, this version defines three different classes named **Student** , **Dog** , and **Cat** that implement the **CallBack** interface. Mixed objects of those three types are registered and maintained on a list and notified at callback time.

As before, this program defines a **CallBack** interface. However, this version of the interface declares two different methods that can be invoked as callback methods instead of just one. In other words, in this case, the objects register to be notified whenever an interesting event from a family of interesting events occurs *(the family has two members)* .

When notified, the objects also need to be advised as to which interesting event actually happened. The methodology for differentiating between the two different kinds of interesting events is to invoke one callback method in the case of one event, and to invoke the other callback method in the case of the other event.

All classes that implement the **CallBack** interface must define both methods, but it is allowable to define the method as an empty method. The net effect of defining a callback method as an empty method is to simply ignore the callback associated with that method. Note however, there is some overhead associated with the invocation of an empty method. *(Although we haven't discussed the topic yet, I believe that this overhead is the reason that JavaSoft chose to separate the **MouseListener** and **MouseMotionListener** interfaces. Perhaps someone will remember to remind me to discuss that when we get to the topic of **MouseMotionListener** interface in the classroom lecture.)*

One of the callback methods in this version of our program is named **recess** and the other callback method is named **lunch** . Thus, registered objects can

be notified either of a *recess* event, or of a *lunch* event.

The **Dog** class ignores the recess callback by defining an empty **recess** method, and the **Cat** class ignores the lunch callback by defining an empty **lunch** method. The **Student** class responds to both types of callbacks with fully- defined methods.

The program defines a **Teacher** class that has the ability to create and maintain a list of objects of the interface type, and to notify those objects that something interesting has happened by invoking either the **recess** method or the **lunch** method on all the objects on the list. It is important to note that every object on the list will be notified of both types of callbacks, although as mentioned above, a particular type of callback can be ignored by a class simply by leaving the callback method empty when it is defined.

As before, objects can be added to the list and then removed from the list. However, removal of objects from the list was demonstrated in the previous program, so removal is not demonstrated in this program.

Notification takes the form of invoking either the **recess** method or the **lunch** method on all the objects on the list.

Finally, the program defines a controlling class that ties all the pieces together and exercises them. The program was originally tested using JDK 1.1.3 under Win95. The program was more recently retested using JDK 8 and Win7. The output from the program is shown later.

Interesting code fragments

The code in Listing 1 defines an interface named **CallBack** that will create a new type and that declares two generic methods that can be used to call back any object that is of a class that implements the interface.

Note: Listing 1 . The CallBack interface.

```
interface CallBack{
  public void recess();
  public void lunch();
}//end interface CallBack
```

Listing 2 defines a class whose objects can maintain a list of registered objects *(registration is the process of placing an object on the list)* of type **CallBack** and can notify all the objects on that list when something interesting happens. This class differs form the one in the earlier lesson in that it has the ability to notify for two different types of callbacks: **recess** and **lunch** .

The name of this class is **Teacher** . The code to construct the **Teacher** object, add objects to the list, and remove objects from the list hasn't changed in a significant way, so we will skip over that code and go straight to the code that is new and different.

Basically what we now have is two different methods in place of one. One of the methods is named **callRecess** and the other is named **callLunch** . Except for their names, these methods are almost identical to the single method named **callTheRoll** in the earlier lesson, so a lot of discussion isn't needed. The code in the method makes a copy of the list and then uses a **for** loop along with some **Vector** methods to access each object reference. Then the callback method is invoked on each object reference. We will show one of the methods below for reference.

Note: Listing 2 . The callRecess method in the Teacher class.

```
  void callRecess(){
    Vector tempList;//save a temporary copy of
list here
```

```
    //Make a copy of the list.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Invoke the recess() method on each object on
    // the list.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{

((CallBack)tempList.elementAt(cnt)).recess();
    }//end for loop
  }//end callRecess()
```

That concludes the discussion of the class named **Teacher** . The **Teacher** class is followed by three class definitions that implement the **CallBack** interface: **Student, Dog** , and **Cat.** These class definitions differ from the ones in the earlier lesson in that they define two callback methods instead of just one: **recess** and **lunch** .

Recall that I said that a class can ignore a particular type of callback simply by defining the callback method as an empty method. Recall also that I said that the **Dog** class ignores the **recess()** callback in just this way.

Because of the similarity of these three classes, I am only going to show one of them below. Listing 3 shows the **Dog** class to illustrate how it defines an empty method to ignore the **recess** callback.

Note: Listing 3 . The Dog class.

```
class Dog implements CallBack{
```

```
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------
-------//

  //An object of the Teacher class can invoke this
method
  // as the callback mechanism.

  public void recess(){//announce recess
    //ignore this callback with an empty method
  }//end overridden recess()
  //-----------------------------------------
-------//

  //An object of the Teacher class can also invoke
this
  // method as a callback mechanism.
  public void lunch(){//announce recess
    System.out.println(name + " lunch");
  }//end overridden lunch()
}//end class Dog
```

That brings us to the controlling class that ties all the pieces together and exercises them *(see Listing 4)* . Except for the fact that the **main** method triggers two callbacks instead of just one, and the code to remove an object from the list was deleted for brevity, this code is essentially the same as the

code in the earlier lesson. Therefore, I am going to delete some of the redundant code from this fragment and primarily show only the new code.

The code in the **main** method of the controlling class instantiates a **Teacher** object named **missJones** , and then instantiates some objects of the three types: **Student** , **Dog** , and **Cat** . These objects are registered for callback by invoking the **register** method on the **Teacher** object.

Then the code triggers a **recess** callback and a **lunch** callback.

Note: Listing 4 . The controlling class named Callback03.

```
class Callback03{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

    //Instantiate some Student objects
    //... code deleted for brevity

    //Instantiate some Dog objects.
    //... code deleted for brevity

    //Instantiate some Cat objects
    //... code deleted for brevity

    //Register some Student, Dog, and Cat objects
with
    // the Teacher object.
    missJones.register(tom);
    missJones.register(spot);
    missJones.register(sue);
    missJones.register(cleo);
    missJones.register(fido);
    missJones.register(peg);
```

```
    missJones.register(kitty);
    missJones.register(bob);
    missJones.register(brownie);

    //Cause the Teacher object to call recess on
all
    // the objects on the list.
    missJones.callRecess();
    //Cause the Teacher object to call lunch on
all
    // the objects on the list.
    missJones.callLunch();
  }//end main()
}//end class Callback03
```

The output produced by this program is shown in Figure 1 .

Note: Figure 1. Output from the program named Callback03.

```
Tom recess
Sue recess
CleoCat recess
Peg recess
KittyKat recess
Bob recess
Tom lunch
SpotDog lunch
Sue lunch
FidoDog lunch
Peg lunch
Bob lunch
BrownieDog lunch
```

Summary

In summary then, we have objects of different classes registered on a common callback list where every object on the list receives a callback for every different type of callback event associated with the list.

The different types of callbacks are established by the method declarations in the **CallBack** interface.

Each class of object that registers for callbacks can either respond to all of the different types of callbacks by providing full definitions for all of the callback methods, or can selectively ignore some types of callbacks by defining those callback methods as empty methods.

Complete program listing

A complete listing of the program named Callback03 is shown in Listing 5 .

Note: Listing 5 . The program named Callback03.

```
/*File Callback03.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a
callback
capability using Interfaces.

This is an enhanced version of the program named
Callback02. You should familiarize yourself with
the earlier program before getting into this
program.


This version defines three different classes named
```

Student, Dog, and Cat that implement the CallBack interface. Mixed objects of those three types are maintained on a list and notified at CallBack time.

As before, this program defines a CallBack interface
that can be used to establish a new type of object.

This version of the interface declares two different methods that can be invoked as callback methods instead of just one. All classes that implement
the interface must define both methods, but it is allowable to define the method as an empty method and
ignore the callback associated with a particular method.

One of the callback methods is now called recess() and
the other is called lunch().

The Dog class ignores the recess() callback by defining
an empty method, and the Cat class ignores the lunch()
callback by defining an empty method. The Student class responds to both types of callbacks with fully-
defined methods.

The program defines a Teacher class that has the ability to create and maintain a list of objects

of the interface type, and to notify those objects that something interesting has happened by

```
invoking
either the recess() method or the lunch() method
on all
the objects on the list.  It is important to note
that
every object on the list will be notified of both
types of callback, although as mentioned above, a
particular type of callback can be ignored simply
by
leaving the method empty when it is defined.

Note that objects can be added to the list and
then
removed from the list.  However, removal of
objects from
the list was demonstrated in the previous program,
so
removal is not demonstrated in this program.

As always, notification takes the form of invoking
either the recess() method or the lunch() method
on all
the objects on the list.

Finally, the program defines a controlling class
that
ties all the pieces together and exercises them.

Tested using JDK 1.1.3 under Win95.

The output from the program was:

Tom recess
Sue recess
CleoCat recess

Peg recess
KittyKat recess
```

```
Bob recess
Tom lunch
SpotDog lunch
Sue lunch
FidoDog lunch
Peg lunch
Bob lunch
BrownieDog lunch
**********************************************
********/
import java.util.*;

//First we define an interface that will create a
new type
// and declare two generic methods that can be
used to
// callback any object that is of a class that
implements
// the interface.
interface CallBack{
  public void recess();
  public void lunch();
}//end interface CallBack
//=================================================
=======//

//Next we need a class whose objects can maintain
a
// registered list of objects of type CallBack and
can
// notify all the objects on that list when
something
// interesting happens. This class has the ability
to
// notify of two different types of callbacks,
recess()
// and lunch().
```

```
class Teacher{
  Vector objList; //list of objects of type
CallBack
  //-----------------------------------------
-------//

  Teacher(){//constructor
    //Instantiate a Vector object to contain the
list
    // of registered objects.
    objList = new Vector();
  }//end constructor
  //-----------------------------------------
-------//

  //Method to add objects to the list.
  synchronized void register(CallBack obj){
    this.objList.addElement(obj);
  }//end register()
  //-----------------------------------------
-------//

  //Method to remove objects from the list.
  synchronized void unRegister(CallBack obj){
    if(this.objList.removeElement(obj))
      System.out.println(obj + " removed");
    else
      System.out.println(obj + " not in the
list");
  }//end register()
  //-----------------------------------------
-------//

  //Method to notify all objects on the list that

  // something interesting has happened regarding
recess.
```

```
  void callRecess(){
    Vector tempList;//save a temporary copy of
list here

    //Make a copy of the list.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Invoke the recess() method on each object on
    // the list.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{

((CallBack)tempList.elementAt(cnt)).recess();
    }//end for loop
  }//end callRecess()
  //-----------------------------------------------
-------//
  //Method to notify all objects on the list that
  // something interesting has happened regarding
lunch.
  void callLunch(){
    Vector tempList;//save a temporary copy of
list here

    //Make a copy of the list.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Invoke the lunch() method on each object on
    // the list.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{

      ((CallBack)tempList.elementAt(cnt)).lunch();
    }//end for loop
```

```
  }//end callRecess()
  //-------------------------------------------
-------//
}//end class Teacher
//=============================================
=======//

//Class that implements the CallBack interface.
Objects
// of this class can be registered on the list
maintained
// by an object of the Teacher class, and will be
notified
// whenever that object invokes either the
recess() method
// or the lunch() method on the registered objects
on
// the list.  This method provides a full
definition for
// both methods.

class Student implements CallBack{
  String name; //store the object name here for
later ID
  //-------------------------------------------
-------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify
the obj
  }//end constructor
  //-------------------------------------------
-------//

  //An object of the Teacher class can invoke this
method
  // as a callback mechanism.
```

```
  public void recess(){//announce recess
    System.out.println(name + " recess");
  }//end overridden recess()
  //-----------------------------------------
-------//

  //An object of the Teacher class can also invoke
this
  // method as a callback mechanism.
  public void lunch(){//announce recess
    System.out.println(name + " lunch");
  }//end overridden lunch()

}//end class Student

//=========================================
=======//

//Another Class that implements the CallBack
interface.
// See description above. This class defines the
recess()
// method as an empty method.

class Dog implements CallBack{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------
-------//


  //An object of the Teacher class can invoke this
```

```
method
  // as the callback mechanism.

  public void recess(){//announce recess
    //ignore this callback with an empty method
  }//end overridden recess()
  //-----------------------------------------
-------//

  //An object of the Teacher class can also invoke
this
  // method as a callback mechanism.
  public void lunch(){//announce recess
    System.out.println(name + " lunch");
  }//end overridden lunch()
}//end class Dog
//=========================================
=======//

//A third Class that implements the CallBack
interface,
// similar to the other two classes.  This class
defines
// the lunch() method as an empty method.

class Cat implements CallBack{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Cat(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------

-------//
```

```
  //An object of the Teacher class can invoke this
method
  // as the callback mechanism.

  public void recess(){//announce recess
    System.out.println(name + " recess");
  }//end overridden recess()
  //-----------------------------------------------
-------//

  //An object of the Teacher class can also invoke
this
  // method as a callback mechanism.
  public void lunch(){//announce recess
    //ignore this callback with an empty method
  }//end overridden lunch()
}//end class Cat
//===============================================
=======//

//Controlling class that ties all the pieces
together and
// exercises them.
class Callback03{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

    //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");


    //Instantiate some Dog objects.
    Dog spot = new Dog("SpotDog");
```

```
    Dog fido = new Dog("FidoDog");
    Dog brownie = new Dog("BrownieDog");

    //Instantiate some Cat objects
    Cat cleo = new Cat("CleoCat");
    Cat kitty = new Cat("KittyKat");

    //Register some Student, Dog, and Cat objects
with
    // the Teacher object.
    missJones.register(tom);
    missJones.register(spot);
    missJones.register(sue);
    missJones.register(cleo);
    missJones.register(fido);
    missJones.register(peg);
    missJones.register(kitty);
    missJones.register(bob);
    missJones.register(brownie);

    //Cause the Teacher object to call recess on
all
    // the objects on the list.
    missJones.callRecess();
    //Cause the Teacher object to call lunch on
all
    // the objects on the list.
    missJones.callLunch();
  }//end main()
}//end class Callback03
//=============================================
=======//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java0078 Java OOP Callbacks - II
- File: Java0078.htm
- Published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0079 Java OOP Callbacks - III
In a real program involving callbacks, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification. In addition, it is often desirable to encapsulate information in an object and to pass that object as a parameter when the callback method is invoked. This lesson will enhance the previous program by providing this capability.

Revised: Wed Jul 06 11:46:37 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Listings
- Introduction
- Sample program
    - Interesting code fragments
        - The interface named CallBack
        - The class named CallBackObjectClass
        - The class named CallBackAdapter
        - The class named Teacher
        - The class named Dog
        - The controlling class named Callback04
- Summary

- Complete program listings
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

This is the third in a series of three lessons on Callbacks in Java. The three lessons are named Callbacks - I, Callbacks - II, and Callbacks - III. Students in Prof. Baldwin's **ITSE 2317 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Listings while you are reading about them.

Listings

- Listing 1 . The interface named CallBack.
- Listing 2 . The class named CallBackObjectClass.
- Listing 3 . The class named CallBackAdapter.
- Listing 4 . Abbreviated version of the Teacher class.
- Listing 5 . The class named Dog.
- Listing 6 . The program named Callback04.

Introduction

Hopefully, by this point, you are familiar with the *callback* material in the earlier lessons. Therefore, I will avoid, insofar as practical, repeating the

introductory material from those lessons.

In a real program involving callbacks, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification.

In addition, it is often desirable to encapsulate information in an object and to pass that object as a parameter when the callback method is invoked. This lesson will enhance the previous program by providing this capability.

Also, sometimes the interfaces used to implement callbacks declare many callback methods, and some classes are interested in only a few of those methods. It can become burdensome to have to provide large numbers of empty methods in all the classes that implement such an interface. For this reason, a mechanism known in Java terms as an **adapter** class is used to reduce the burden.

This lesson will show you how to create and use such **adapter** classes.

We will continue with our earlier scenario consisting of a *teacher* and some *students* . At this point, we have expanded our scenario to include many students and some animals in the classroom.

The students *(and the animals)* register themselves on the teachers roll book to be notified when it is time for recess, or it is time for lunch.

In this lesson, as in the earlier lesson, all of the students and all of the animals receive notification of both types of events (*recess* or *lunch*) but some of those who are notified choose to ignore the notification. However, in this lesson, the methodology for ignoring the notification is different than it was in the previous lesson.

Sample program

In case you just started reading at this point, this program named **Callback04** is an enhanced version of the earlier program named

Callback03 . You should familiarize yourself with the earlier program before trying to understand this program.

This version of the program makes two unrelated enhancements to the program named **Callback03** .

- First, a new class is defined in conjunction with the **CallBack** interface that makes it possible for the **Teacher** class to encapsulate information into an object and pass that object as a parameter whenever a callback is made.
- Second, a new class is defined in conjunction with the **CallBack** interface which, in Java terminology, is often called an **adapter** class.

An **adapter** class, as defined here, is a convenience class that implements the interface and defines all the methods declared in the interface with empty methods. Then any class that needs to implement the interface can ***extend*** the adapter class without a requirement to define those methods that are not of interest.

In this version of the program, the **CallBack** interface is expanded to declare several dummy methods to emphasize this benefit of the use of an **adapter** class.

Note that objects of a class that *extends* the adapter class can be referred to by the interface type. Thus, interface type is an inherited attribute that results from *extending* a class that *implements* an interface.

Using an **adapter** class also makes it possible to expand the interface later, by declaring new methods, without breaking code that already **implements** the interface, provided that the **adapter** class is appropriately redefined. In this sense, the **adapter** class is a little more than a convenience class.

This version of the program defines three different classes named **Student** , **Dog** , and **Cat** , that *implement* the **CallBack** interface *indirectly* by extending the adapter class.

Mixed objects of these three types are maintained in a common list and notified at callback time.

As in the previous version of the program, the **Dog** class ignores the **recess** callback and the **Cat** class ignores the **lunch** callback. In this program, these classes simply don't bother to override methods of the interface that are not of interest.

The **Student** class responds to both types of callbacks with fully- defined methods.

The program defines a **Teacher** class that has the ability to create and maintain a list of objects of the interface type, and to notify those objects that something interesting has happened by invoking either the **recess** method or the **lunch** method on all the objects on the list. An object is instantiated and passed as a parameter whenever one or the other of these methods is invoked.

Finally, the program defines a controlling class that ties all the pieces together and exercises them.

The program was originally tested using JDK 1.1.3 under Win95. More recently it was tested using Java 8 and Windows 7.

Interesting code fragments

The interface named CallBack

We begin by defining an interface named **CallBack** that will create a new type and will declare two generic methods that can be used to callback any object that is of a class that implements the interface. The interface named **CallBack** is shown in Listing 1 .

Note: Listing 1 . The interface named CallBack.

```
interface CallBack{
  public void recess(CallBackObjectClass obj);
  public void lunch(CallBackObjectClass obj);

  public void dummy1(CallBackObjectClass obj);
  public void dummy2(CallBackObjectClass obj);
  public void dummy3(CallBackObjectClass obj);
  public void dummy4(CallBackObjectClass obj);
  public void dummy5(CallBackObjectClass obj);
  public void dummy6(CallBackObjectClass obj);

}//end interface CallBack
```

Note that in addition to two methods mentioned above, the interface also declares several additional dummy methods to emphasize the benefit of extending the adapter class that is defined later.

Note also that the methods now require a parameter, which was not the case in earlier lessons. The class for this parameter will be defined soon next.

The class named CallBackObjectClass

Listing 2 defines a class from which we can instantiate an object to pass as a parameter in the callback methods. This class can be as simple or as elaborate as needed to convey the amount and type of information needed with the callback. In our case, it contains a single **String** object.

Note: Listing 2 . The class named CallBackObjectClass.

```
class CallBackObjectClass{
```

```
  String data;

  CallBackObjectClass(String data){//constructor
    this.data = data;
  }//end constructor
}//end CallBackObjectClass
```

The class named CallBackAdapter

Listing 3 shows the definition of the adapter class for the **CallBack** interface. This class defines all the interface methods with empty methods.

Note: Listing 3 . The class named CallBackAdapter.

```
class CallBackAdapter implements CallBack{
  public void recess(CallBackObjectClass obj){};
  public void lunch(CallBackObjectClass obj){};

  public void dummy1(CallBackObjectClass obj){};
  public void dummy2(CallBackObjectClass obj){};
  public void dummy3(CallBackObjectClass obj){};
  public void dummy4(CallBackObjectClass obj){};
  public void dummy5(CallBackObjectClass obj){};
  public void dummy6(CallBackObjectClass obj){};
}//end class CallBackAdapter
```

Classes that need to *implement* the **CallBack** interface can now

- *extend* the **CallBackAdapter** class,
- *override* only the empty methods of interest, and

- ignore the other methods.

It is important to note that the existence of the **adapter** does not prevent another class from *implementing* the interface instead of *extending* the **adapter** class.

Objects of a class that *extends* the **CallBackAdapter** class can be referenced as the interface type **CallBack** .

The class named Teacher

Next we need a class whose objects can maintain a list of registered objects of type **CallBack** and can notify all the objects on that list when something interesting happens. This class has the ability to invoke two different callback methods: **recess** and **lunch** .

Obviously, if you have been following this development, you will know by now that this is the class named **Teacher** .

The only thing significantly different about this version of the **Teacher** class is the fact that it instantiates and passes an object of type **CallBackObjectClass** whenever it makes a callback.

Since you have seen this method several times before, and should be very familiar with it by now, I am going to delete most of the code in the class and show you only that portion of the code that invokes the **recess** callback method on all the objects on the list, instantiating and passing an object of type **CallBackObjectClass** as a parameter in each case. The abbreviated version of the **Teacher** class is shown in Listing 4 .

Note: Listing 4 . Abbreviated version of the Teacher class.

```
class Teacher{
  //code deleted
```

```
  void callRecess(){
    //code deleted

    for(int cnt = 0; cnt < tempList.size(); cnt++)
{
      ((CallBack)tempList.elementAt(cnt)).recess(
                        new CallBackObjectClass("
Recess"));
    }//end for loop
  }//end callRecess()
  //code deleted

}//end class Teacher
```

The class named Dog

That brings us to the three classes that implement the **CallBack** interface *indirectly* by *extending* the **CallBackAdapter** class.

Objects of these three classes can be registered on the list maintained by an object of the **Teacher** class, and will be notified whenever that object invokes either the **recess** method or the **lunch** method on the registered objects on the list.

Part of the notification will be an object passed as a parameter to the callback method. Code in the callback method can either ignore that object, or can extract the information from the object and use it.

These three classes are very similar, so I will show you only one of them here. To be consistent with the previous lesson, I will show you the definition for the **Dog** class that ignores the **recess** callback. This class, which is shown in Listing 5 , also ignores all of the *dummy* methods that were declared in the **CallBack** i interface in this program.

Note: Listing 5 . The class named Dog.

```
class Dog extends CallBackAdapter{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------
-------//

  //An object of the Teacher class can invoke this
  // method as a callback mechanism passing an
object
  // as a parameter.
  public void lunch(CallBackObjectClass obj)
{//announce
    System.out.println(name + obj.data);
  }//end overridden lunch()
  //-----------------------------------------
-------//
}//end class Dog
```

Note first that this class *extends* the **CallBackAdapter** class and does not *implement* any interfaces.

Note also that it only overrides the **lunch** method from the **adapter** class, ignoring all of the other empty methods defined in that class. Were it not for the **adapter** class in this case, this **Dog** class definition would have to define the six dummy methods and the **recess** method with empty methods.

Therefore, the availability of an adapter class can be a real convenience for the programmer.

Finally, note that code in the **lunch** method extracts the instance variable named **data** and displays that information instead of a literal string as was the case in the earlier versions of this program.

The controlling class named Callback04

That brings us to the controlling class that ties all the other pieces together and exercises them. The controlling class in this version of the program is no different from that of the earlier version named **Callback03** .

Also, the output from running this version of the program is almost the same as the earlier version, with the main difference being that the callback methods display information encapsulated in an object passed to the method which was not the case in the earlier versions. The output can be seen in the complete program listing shown in Listing 6 .

Therefore, there should be no need to provide any further discussion regarding the testing of this version of the program.

Summary

We have objects of different classes registered on a common callback list where every object on the list receives a callback for every different type of callback event associated with the list.

Whenever a callback occurs, an object is instantiated to contain information and passed as a parameter to the callback method. The code in the callback method can either ignore that object, or extract the information from the object and use it. In this program, the code in the callback methods extracts and displays the information encapsulated in the object.

The different types of callbacks are established by the method declarations in the **CallBack** interface.

An **adapter** class is defined that implements the **CallBack** interface and defines all of its methods as empty methods.

Each class of object that registers for callbacks can either respond to all of the different types of callbacks by overriding all of the callback methods in the **adapter** class, or can selectively ignore some types of callbacks simply by not overriding those methods.

Such a class can also ignore the **adapter** class and *implement* the interface directly as in the earlier lessons.

Complete program listings

A complete listing of the program is shown in Listing 6 .

Note: Listing 6 . The program named Callback04.

```
/*File Callback04.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a
callback
capability using Interfaces.

This is an enhanced version of the program named
Callback03.  You should familiarize yourself with
the earlier program before getting into this
program.

This version of the program makes two unrelated
enhancements to the program named Callback03.

First, a new class is defined in conjunction with
the

CallBack interface that makes it possible for the
Teacher class to encapsulate information into an
```

```
object
and pass that object whenever a callback is made.

Second, a new class is defined in conjunction with
the
CallBack interface which, in Java terminology, is
often
called an adapter class.  This is a convenience
class
that implements the interface and defines all the
methods declared in the interface with empty
methods.
Then any class that needs to implement the
interface
can extend the adapter class without a requirement
to
define those methods that are not of interest.

In this case, the interface is expanded to declare
several dummy methods to emphasize this benefit of
the use of an adapter class.

Note that objects of classes that extend the
adapter
class that implements the interface can be
referred to
by the interface type. Thus, interface type is an
inherited attribute that results from extending a
class.

Using an adapter class also makes it possible to
expand
the interface later by declaring new methods
without
breaking code that already implements the
interface.
```

```
This version defines three different classes named
Student, Dog, and Cat, that implement the CallBack
interface indirectly by extending the adapter
class.
These classes simply don't bother to define
methods of the interface that are not of interest.

Mixed objects of these three types are maintained
on a list and notified at CallBack time.

The Dog class ignores the recess() callback and
the
Cat class ignores the lunch() callback.  The
Student
class responds to both types of callbacks with
fully-
defined methods.

The program defines a Teacher class that has the
ability to create and maintain a list of objects
of the interface type, and to notify those objects
that something interesting has happened by
invoking
either the recess() method or the lunch() method
on all
the objects on the list.

Finally, the program defines a controlling class
that
ties all the pieces together and exercises them.

Tested using JDK 1.1.3 under Win95.

The output from the program was:


Tom Recess
Sue Recess
```

```
CleoCat Recess
Peg Recess
KittyKat Recess
Bob Recess
Tom Lunch
SpotDog Lunch
Sue Lunch
FidoDog Lunch
Peg Lunch
Bob Lunch
BrownieDog Lunch
**********************************************
********/
import java.util.*;

//First we define an interface that will create a
new type
// and declare two generic methods that can be
used to
// callback any object that is of a class that
implements
// the interface. Note that the methods now
require a
// parameter. Note also that we have declared
several
// additional dummy methods to emphasize the
benefit of
// extending the adapter class that is defined
later.
interface CallBack{
  public void recess(CallBackObjectClass obj);
  public void lunch(CallBackObjectClass obj);

  public void dummy1(CallBackObjectClass obj);
  public void dummy2(CallBackObjectClass obj);

  public void dummy3(CallBackObjectClass obj);
  public void dummy4(CallBackObjectClass obj);
```

```
  public void dummy5(CallBackObjectClass obj);
  public void dummy6(CallBackObjectClass obj);

}//end interface CallBack
//===========================================
=======//

//Now we need a class that can be instantiated to
// pass an object in the callback methods.
class CallBackObjectClass{
  String data;

  CallBackObjectClass(String data){//constructor
    this.data = data;
  }//end constructor
}//end CallBackObjectClass
//===========================================
=======//

//Now we need an adapter class for the CallBack
interface.
// This class defines all the interface methods
with
// empty methods.  Classes that need the interface
can
// now extend this class and override only the
empty
// methods of interest, and ignore the others.
Note that
// objects of a class that extends this class can
be
// referenced as the interface type CallBack.
class CallBackAdapter implements CallBack{
  public void recess(CallBackObjectClass obj){};
  public void lunch(CallBackObjectClass obj){};


  public void dummy1(CallBackObjectClass obj){};
```

```
  public void dummy2(CallBackObjectClass obj){};
  public void dummy3(CallBackObjectClass obj){};
  public void dummy4(CallBackObjectClass obj){};
  public void dummy5(CallBackObjectClass obj){};
  public void dummy6(CallBackObjectClass obj){};
}//end class CallBackAdapter
//===========================================
=======//

//Next we need a class whose objects can maintain
a
// registered list of objects of type CallBack and
can
// notify all the objects on that list when
something
// interesting happens. This class has the ability
to
// notify two different types of callbacks,
recess()
// and lunch().

class Teacher{
  Vector objList; //list of objects of type
CallBack
  //-----------------------------------------
-------//

  Teacher(){//constructor
    //Instantiate a Vector object to contain the
list
    // of registered objects.
    objList = new Vector();
  }//end constructor
  //-----------------------------------------
-------//


  //Method to add objects to the list.
```

```
  synchronized void register(CallBack obj){
    this.objList.addElement(obj);
  }//end register()
  //-----------------------------------------
-------//

  //Method to remove objects from the list.
  synchronized void unRegister(CallBack obj){
    if(this.objList.removeElement(obj))
      System.out.println(obj + " removed");
    else
      System.out.println(obj + " not in the
list");
  }//end register()
  //-----------------------------------------
-------//

  //Method to notify all objects on the list that
  // something interesting has happened regarding
  // recess and to pass information to the object
  // encapsulated in an object as a parameter.
  void callRecess(){
    Vector tempList;//save a temporary copy of
list here

    //Make a copy of the list.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Invoke the recess() method on each object on
    // the list, passing an object as a parameter.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{
      ((CallBack)tempList.elementAt(cnt)).recess(
                        new CallBackObjectClass("
Recess"));
```

```
    }//end for loop
  }//end callRecess()
  //-------------------------------------------
-------//
  //Method to notify all objects on the list that
  // something interesting has happened regarding
  // lunch and to pass an object containing
information
  // as a parameter to the callback method.
  void callLunch(){
    Vector tempList;//save a temporary copy of
list here

    //Make a copy of the list.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Invoke the lunch() method on each object on
    // the list, passing an object as a parameter.
    for(int cnt = 0; cnt < tempList.size(); cnt++)
{
      ((CallBack)tempList.elementAt(cnt)).lunch(
                        new CallBackObjectClass("
Lunch"));
    }//end for loop
  }//end callRecess()
  //-------------------------------------------
-------//
}//end class Teacher
//=============================================
=======//

//Class that implements the CallBack interface
indirectly

// by extending the CallBackAdapter class.
Objects
```

```
// of this class can be registered on the list
maintained
// by an object of the Teacher class, and will be
notified
// whenever that object invokes either the
recess() method
// or the lunch() method on the registered objects
on
// the list.  This method provides a full
definition for
// both methods.

class Student extends CallBackAdapter{
  String name; //store the object name here for
later ID
  //-----------------------------------------------
-------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------------
-------//

  //An object of the Teacher class can invoke this
method
  // as a callback mechanism. Note that this
method
  // displays the data encapsulated in the
incoming
  // object.
  public void recess(CallBackObjectClass obj)
{//announce
    System.out.println(name + obj.data);

  }//end overridden recess()
  //-----------------------------------------------
```

```
-------//

  //An object of the Teacher class can also invoke
this
  // method as a callback mechanism passing an
object
  // as a parameter.
  public void lunch(CallBackObjectClass obj)
{//announce
    System.out.println(name + obj.data);
  }//end overridden lunch()
  //-----------------------------------------
-------//
}//end class Student

//=========================================
=======//

//Another Class that implements the CallBack
interface
// indirectly by extending the CallBackAdapter
class.
// See description above. This class ignores the
recess()
// method as well as the dummy methods.

class Dog extends CallBackAdapter{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor

  //-----------------------------------------
-------//
```

```
  //An object of the Teacher class can invoke this
  // method as a callback mechanism passing an
object
  // as a parameter.
  public void lunch(CallBackObjectClass obj)
{//announce
    System.out.println(name + obj.data);
  }//end overridden lunch()
  //-----------------------------------------
-------//
}//end class Dog
//=============================================
=======//

//A third Class that implements the CallBack
interface
// indirectly by extending the CallBackAdapter
class,
// similar to the other two classes.  This class
ignores
// the lunch() method as well as the dummy
methods.

class Cat extends CallBackAdapter{
  String name; //store name here for later ID
  //-----------------------------------------
-------//

  Cat(String name){//constructor
    this.name = name; //save the name to identify
the obj
  }//end constructor
  //-----------------------------------------
-------//


  //An object of the Teacher class can invoke this
```

```
method
  // as the callback mechanism, passing an object
as a
  // parameter.
  public void recess(CallBackObjectClass obj)
{//announce
    System.out.println(name + obj.data);
  }//end overridden recess()
  //-----------------------------------------------
-------//
}//end class Cat
//=================================================
=======//

//Controlling class that ties all the pieces
together and
// exercises them.
class Callback04{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

    //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");

    //Instantiate some Dog objects.
    Dog spot = new Dog("SpotDog");
    Dog fido = new Dog("FidoDog");
    Dog brownie = new Dog("BrownieDog");

    //Instantiate some Cat objects

    Cat cleo = new Cat("CleoCat");
    Cat kitty = new Cat("KittyKat");
```

```
    //Register some Student, Dog, and Cat objects
with
    // the Teacher object.
    missJones.register(tom);
    missJones.register(spot);
    missJones.register(sue);
    missJones.register(cleo);
    missJones.register(fido);
    missJones.register(peg);
    missJones.register(kitty);
    missJones.register(bob);
    missJones.register(brownie);

    //Cause the Teacher object to call recess on
all
    // the objects on the list.
    missJones.callRecess();
    //Cause the Teacher object to call lunch on
all
    // the objects on the list.
    missJones.callLunch();
  }//end main()
}//end class Callback04
//=============================================
=======//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java0079 Java OOP Callbacks - III
- File: Java0079.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0080 Java OOP Event Handling in JDK 1.1, A First Look, Delegation Event Model
This is the first module involving the use of the Delegation Event Model.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- Design Goals of the JDK 1.1 Delegation Event Model
- Simplified Overview of the New Delegation Model
    - Sample Program
    - Interesting Code Fragments
    - Program Listing for Event08
- More Detailed Overview of the Delegation Model
    - Event Hierarchy
    - Low-level vs. Semantic Events
    - Event Listeners
    - Event sources
    - Adapters
    - Filtering for Performance
- Another Sample Program
    - Event09 Interesting Code Fragments
    - Event09 Program Listing
- Summary
- Review
- Run the program

- Summary
- Miscellaneous

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

(Editor's note: As you read this module, you will see that it was originally written in 1997. However, despite many improvements in Java since 1997, most of what was true in 1997 is still true in 2012.)

This module makes several references to my website, which is located at http://www.dickbaldwin.com/toc.htm . A copy of the original html version of this module is available here .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output.
- Figure 2 . Screen output from Event09.

Listings

- Listing 1 . The main method for Event08.
- Listing 2 . Beginning of the constructor.
- Listing 3 . Instantiate two Listener objects.
- Listing 4 . Register the listener objects.

- Listing 5 . Make the Frame visible.
- Listing 6 . Beginning of the class named WProc1.
- Listing 7 . The windowClosing event handler.
- Listing 8 . Definition for the class named WProc2.
- Listing 9 . Complete listing for Event08.
- Listing 10 . Beginning of the class named Event09.
- Listing 11 . The class named MyFrame.
- Listing 12 . Beginning of the class named GUI.
- Listing 13 . Register two listener objects.
- Listing 14 . The class named MouseProc.
- Listing 15 . The class named WProc1.
- Listing 16 . Complete listing of Event09.

Introduction

This module provides a description of the Delegation Event Model as defined in JDK 1.1 along with sample programs that illustrate some aspects of the model. I describe how the event model maps to the AWT API.

Much of this information was taken from the documentation released with the various versions of JDK 1.1, and is the intellectual property of Sun Microsystems. The material is being reproduced here for the sole purpose of assisting students in learning how to use the event model.

Design Goals of the JDK 1.1 Delegation Model

According to Sun, the primary design goals of the event model in the AWT are the following:

Note: Design goals:

- Simple and easy to learn
- Support a clean separation between application and GUI code
- Facilitate the creation of robust event handling code which is less error-prone (strong compile-time checking)

- Flexible enough to enable varied application models for event flow and propagation
- For visual tool builders, enable run-time discovery of both events that a component generates as well as the events it may observe
- Support backward binary compatibility with the old model

It will be up to you to decide if they have met their goals.

Simplified Overview of the New Delegation Model

First, I will provide a simplified overview of the model including a sample program. Following that, I will embark on a detailed discussion of the model. Hopefully the simplified discussion and the sample program will help you to understand the material in the detailed discussion.

Also, the earlier module on callbacks should have given you considerable background to help you understand this material. If you are not familiar with the callback material, you should go back and review that material.

Event sources and listeners

Events are now organized into a hierarchy of event classes. The model makes use of **event sources** and **event listeners** .

An event **source** is an object that has the ability

- to determine when an interesting event has occurred, and
- to notify listener objects of the occurrence of the event.

Although you as the programmer establish the framework for such notification, the actual notification takes place automatically behind the scenes.

A **listener** object is an instance of a class *(or instance of a subclass of a class)* that

- implements a specific listener interface.

A number of listener interfaces are defined where each interface declares the methods appropriate for a specific class of events. Thus, there is natural pairing of classes of events and interface definitions.

For example, there is a class of mouse events that includes most of the events normally associated with mouse action. There is a matching interface definition, which is used to define a listener class for those events *(actually this is the one case where two interfaces are defined to match up with a single event class)* .

Registration

A listener object can be *registered* on a source object to be notified of the occurrence of all events of the specific class for which the listener object is designed.

Once a listener object is registered to be notified of those events, the occurrence of an event defined by the specified class will automatically call the matching method in the listener object. The code in the body of the method is designed by the programmer to perform the desired action when the event occurs.

Multiple notification methods

Some event classes *(such as the mouse events, for example)* involve a number of different possible event types *(notification methods)* . A listener class that implements the matching interface for that event class must implement or define *(provide a body for)* all the methods declared in the interface.

Adapter classes

An intermediate set of classes, known as *Adapter* classes, is provided to prevent this from becoming burdensome. These Adapter classes implement the listener interfaces, and define the matching interface methods with empty methods.

A listener class can then be defined that *extends* the Adapter class instead of *implementing* the corresponding listener interface. With this approach, the listener class need only override those methods of interest since the requirement to define all of the interface methods has already been satisfied by the Adapter class.

An example

For example, in the sample program that follows, two different listener objects instantiated from two different listener classes are registered to receive all events involving the manipulation of a **Frame** object *(opening, closing, minimizing, etc.)* .

One of the listener classes *implements* the **WindowListener** interface and hence must define all six methods of that interface.

The other listener class *extends* the **WindowAdapter** class, which in turn *implements* the **WindowListener** interface. The **WindowAdapter** class defines all six methods as empty methods. Thus, this listener class can get by with overriding only two of the six methods of the interface.

Sample Program

This program named Event08 was designed for simplicity. In this program, the code in each of the methods is simple. In all cases but one, the code simply displays a message indicating that the method has been called. Obviously, in order for a program to be of much value, the body of code in the methods would have to be much more substantive, or at least would have to call other methods that are more substantive.

This program illustrates the use of *Event Sources* , *Event Listeners* , and *Adapters* in the *Delegation Event Model* .

Description

Briefly, this program instantiates an object that creates a user interface consisting of a simple **Frame** object. This object is an *Event Source* that

notifies two different *Event Listener* objects of **Window** events.

One of the **Listener** objects *implements* the **WindowListener** interface and defines all of the methods declared in that interface. The other **Listener** object *extends* the **Adapter** class named **WindowAdapter** .

As explained earlier, the purpose of **Adapter** classes *(as used in this context)* is to implement the **Listener** interfaces and to define all of the methods with empty methods.

Classes that *extend* the **Adapter** classes can then selectively override only those methods of interest. This **Listener** object overrides only two of the methods.

(Note that this program does not terminate and return control to the operating system. You must forcefully terminate it.)

This program was originally tested using JDK 1.1.3 under Win95. More recently it was tested using JDK 1.7 under Windows Vista.

The output produced by running the program is presented later in this module.

Interesting Code Fragments

Listing 1 shows the **main** method of the controlling class that instantiates a Graphical User Interface (GUI) object. There are more compact *(and more cryptic)* ways to accomplish this objective, but for the time being, in order to achieve clarity, I will use this approach.

Listing 1 - The main method for Event08.

Listing 1 - The main method for Event08.

```
public class Event08 {//controlling class
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
}//end class Event08
```

The **GUI** class is rather long, so I will break it up and discuss it in parts.

This class is used

- to instantiate and display a user interface object,
- to instantiate two **Listener** objects, and
- to register those two objects for notification whenever a **Window** event occurs.

The class begins by defining its own constructor. The first interesting code in the constructor is the code to instantiate an object of type **Frame** , set its size, and give it a title, as shown in Listing 2 .

Listing 2 - Beginning of the constructor.

Listing 2 - Beginning of the constructor.

```
class GUI{
  public GUI(){//constructor
    //Create a new Frame object
    Frame displayWindow = new Frame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Copyright 1997,
R.G.Baldwin");
```

What is a Frame?

The JDK documentation describes a **Frame** as follows:

Note: Description of a Frame:
A **Frame** is a top-level window with a title and a border. The default layout for a frame is **BorderLayout** . *(We will learn more about layout managers later)*
Frames are capable of generating the following types of window events:

- WindowOpened
- WindowClosing
- WindowClosed
- WindowIconified
- WindowDeiconified
- WindowActivated
- WindowDeactivated.

Stated differently, a **Frame** object is the type of **GUI** object that we might refer to as a window or a form in a typical **GUI** environment.

Instantiate two Listener objects

Listing 3 instantiates two **Listener** objects that will process **Window** events.

The class definitions, named **Wproc1** and **Wproc2** , for these two classes will follow the discussion of the **GUI** class.

*(Note that I am passing a reference to the **Frame** object to the constructor for one of these classes. There is a better (and more cryptic) way to achieve the same objective without the need to pass the reference. Again, since this program was designed for clarity, I am doing it the obvious way.)*

Listing 3 - Instantiate two Listener objects.

```
    WProc1 winProcCmd1 = new
WProc1(displayWindow);
    WProc2 winProcCmd2 = new WProc2();
```

Register the listener objects

The code in Listing 4 is extremely important. This is the code by which we register the listener objects for notification of **Window** events where the **Frame** object named **displayWindow** is the source of the events.

Pay close attention to the syntax of these two statements, because you will be using this syntax often. In subsequent modules, we will dig deeper into the *Delegation Event Model* and I will teach you what is going on behind the scenes when you execute statements of this sort.

Listing 4 - Register the listener objects .

```
displayWindow.addWindowListener(winProcCmd1);

displayWindow.addWindowListener(winProcCmd2);
```

Interpretation

The interpretation of the code in Listing 4 is that the two listener objects named **winProcCmd1** and **winProcCmd2** are added to a list of listener objects that are to be automatically notified

- whenever an event of the **WindowEvent** class occurs
- with respect to the **Frame** object named **displayWindow** .

These listener objects are notified by calling the methods in the objects that match the specific type of the event *(open window, close window, closing window, minimize window, etc.)* .

Make the Frame visible

We wrap up the definition of our **GUI** class with the statement shown in Listing 5 that causes the **Frame** object to become visible on the screen. Note that this statement also causes the **windowActivated** and **windowOpened** events to be generated. *(Current jargon would say that those events are "fired".)*

Listing 5 - Make the Frame visible.

Listing 5 - Make the Frame visible.

```
    displayWindow.setVisible(true);
  }//end constructor
}//end GUI class definition
```

Code for the Listener objects

At this point, we have examined the code that will create a **Frame** object and display it on the screen. Also we have examined the code that *registers* two **Listener** objects on the **Frame** object. However, at this point, the program cannot be compiled because the classes from which the two **Listener** objects are to be instantiated have not yet been defined. That will be our next assignment.

Beginning of the class named WProc1

Listing 6 begins the definition of the classes that can be used to instantiate **Listener** objects.

The first class definition that we will look at *implements* the **WindowListener** interface. This requires that all the methods declared in the interface be defined in this class. This class defines all of the methods. Each of the methods displays a descriptive message whenever it is called.

I will begin the discussion with the constructor as shown in Listing 6 .

Listing 6 - Beginning of the class named WProc1.

Listing 6 - Beginning of the class named WProc1.

```
class WProc1 implements WindowListener{
  Frame displayWindowRef;

  WProc1(Frame windowIn){//constructor
    this.displayWindowRef = windowIn;
  }//end constructor
```

*(Note that the constructor for this class requires a reference to the **Source** object to be passed as a parameter. As mentioned earlier, this is not the preferred way to accomplish our objective, but it is the most straightforward and easy to understand. That is why I elected to use it here. The code in the constructor saves a reference to the **Source** object in an instance variable named **displayWindowRef** .)*

Implementing the WindowListener interface

It is important to note that this class *implements* the **WindowListener** interface. This means that it must define all of the methods that are declared in that interface.

If you examine the complete program listing that is presented later, you will see that there is one method definition in this class definition for each of the methods that are declared in the **WindowListener** interface.

The definition of all the interface methods are very similar, so I am not going to show all of them here. However, I will show the one that requires the reference to the source object that was discussed above, which is the most complicated one of the set.

The windowClosing event handler

The methods that are declared in the listener interfaces are often referred to as *event handler* methods or *event handlers* for short. The **windowClosing**

event handler method is shown in Listing 7 . As you can see, this method starts out by displaying a message. Then it calls the **dispose** method on the **Frame** object. That is the reason that a reference to the **Frame** object was required to be passed in as a parameter.

Calling the **dispose** method causes a **WindowClosed** event to be fired. *(As far as I know, that is the only way to cause a WindowClosed event to be fired.)*

Listing 7 - The windowClosing event handler.

```
  public void windowClosing(WindowEvent e){
    System.out.println("WProc1 windowClosing
test msg");
    displayWindowRef.dispose();//generate
WindowClosed
  }//end windowClosing()
```

The class named WProc2

The class definition shown in Listing 8 does not *implement* the **WindowListener** interface. Instead, it *extends* the **WindowAdapter** class. Therefore, it can selectively override only those methods that are of interest.

In this case, only two of the methods of the **WindowListener** interface are overridden. These overridden methods display a message whenever they are called.

Listing 8 - Definition for the class named WProc2.

```
class WProc2 extends WindowAdapter{

  public void windowIconified(WindowEvent e){
    System.out.println(
              "******** WProc2 windowIconified
test msg");
  }//end windowIconified()

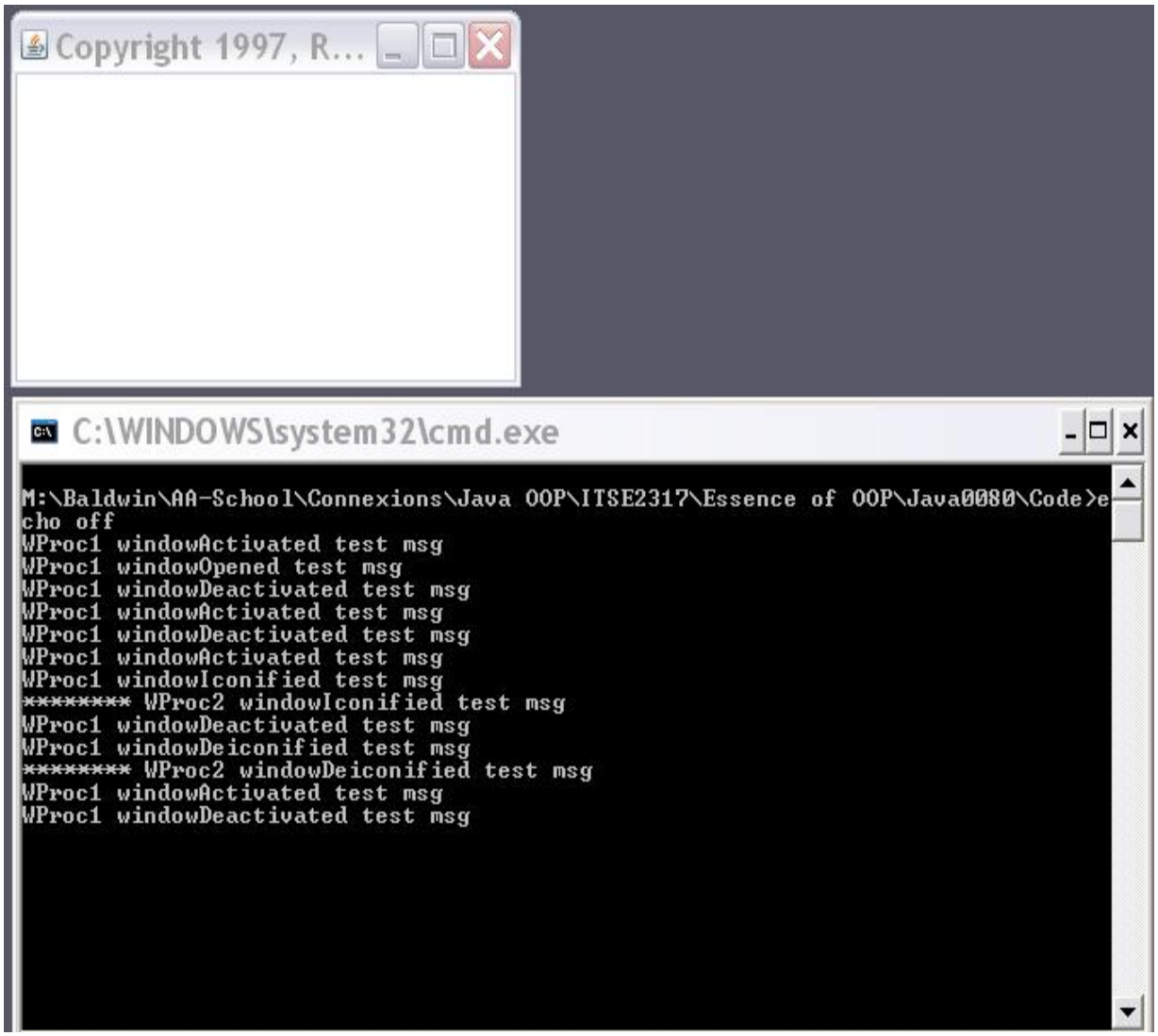
  public void windowDeiconified(WindowEvent e)
{
    System.out.println(
             "******** WProc2 windowDeiconified
test msg");
  }//end windowDeiconified()

}//end class WProc2
```

Program output

Figure 1 shows a sample output from this program. The small **Frame** in the upper-left corner is the graphic **Frame** object. A command-line screen with text output is shown immediately below the Frame.

Figure 1 - Program output.



The text output

The output from running this program for a variety of user actions is shown below. You should be able to correlate the messages shown in the output with the event handler methods discussed above and the user actions that caused those event handlers to be called.

Note: Text output from Event08

```
This program was tested using JDK 1.1.3 under
Win95.
```

```
When executed, this application places a simple
empty
Frame object on the screen.

Starting the program produces the following
output:
WProc1 windowActivated test msg
Wproc1 windowOpened test msg

Pressing the minimize button on the Frame produces
the
following output:

WProc1 windowIconified test msg
******** WProc2 windowIconified test msg
WProc1 windowDeactivated test msg

Restoring the Frame after minimization produces
the
following output:

WProc1 windowActivated test msg
WProc1 windowDeiconified test msg
******** WProc2 windowDeiconified test msg
WProc1 windowActivated test msg

Closing the Frame by pressing the X-icon in the
upper
right of the Frame produces the following output.

WProc1 windowClosing test msg
WProc1 windowDeactivated test msg
WProc1 windowClosed test msg
```

Program Listing for Event08

A complete listing of the program is provided in Listing 9 . You can view the code that was not shown above in this listing.

Listing 9 - Complete listing for Event08.

```
/*File Event08.java Copyright 1997, R.G.Baldwin
Rev 01/10/98

Illustrates the use of Event Sources, Event
Listeners, and
Adapters in the Delegation Event Model.

Briefly, this application instantiates an object
that
creates a user interface consisting of a simple
Frame
object.  This object is an Event Source that
notifies two
different Event Listener objects of Window events.

One of the Listener objects implements the
WindowListener
interface and overrides all of the methods
declared in
that interface.

The other Listener object extends the Adapter
class named
WindowAdapter.  The purpose of Adapter classes is
to
implement the  Listener interfaces and to define
all of
the methods with empty methods.  Classes that
extend the
Adapter classes can then selectively override only
```

those
methods of interest. This Listener object overrides only
two of the methods.

Note that this application does not terminate and return
control to the operating system. You must forcefully
terminate it.

This program was tested using JDK 1.1.3 under Win95.

When executed, this application places a simple empty
Frame object on the screen.

Starting the program produces the following output:
WProc1 windowActivated test msg
Wproc1 windowOpened test msg

Pressing the minimize button on the Frame produces the
following output:

WProc1 windowIconified test msg
******** WProc2 windowIconified test msg
WProc1 windowDeactivated test msg

Restoring the Frame after minimization produces the
following output:

WProc1 windowActivated test msg
WProc1 windowDeiconified test msg

```
******** WProc2 windowDeiconified test msg
WProc1 windowActivated test msg

Closing the Frame by pressing the X-icon in the
upper
right of the Frame produces the following output.

WProc1 windowClosing test msg
WProc1 windowDeactivated test msg
WProc1 windowClosed test msg
**********************************************
********/

import java.awt.*;
import java.awt.event.*;

public class Event08 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
}//end class Event08
//=============================================
=======//

//The following class is used to instantiate a
user
// interface object, to instantiate two Listener
objects,
// and to register those two objects for
notification
// whenever a Window event occurs.
class GUI{
  public GUI(){//constructor
    //Create a new Frame object
    Frame displayWindow = new Frame();
    displayWindow.setSize(300,200);
```

```
    displayWindow.setTitle("Copyright 1997,
R.G.Baldwin");

    //Instantiate two Listener objects that will
process
    // Window events
    WProc1 winProcCmd1 = new
WProc1(displayWindow);
    WProc2 winProcCmd2 = new WProc2();

    //Register the Listener objects for
notification of
    // Window events. This object is the Event
Source.
    displayWindow.addWindowListener(winProcCmd1);
    displayWindow.addWindowListener(winProcCmd2);

    //windowActivated and windowOpened test
messages
    // are produced here
    displayWindow.setVisible(true);

  }//end constructor
}//end class GUI definition
//=============================================
=======//

//The following two classes can be used to
instantiate
// Listener objects. Note that this class
implements the
// WindowListener interface.  This requires that
all the
// methods declared in the interface be overridden
in this
// class. This class overrides all of the methods
and
```

```
// displays a descriptive message whenever one of
the
// methods is called.
class WProc1 implements WindowListener{
  //used to save a reference to the Frame object
  Frame displayWindowRef;

  WProc1(Frame windowIn){//constructor
    // save ref to Frame object
    this.displayWindowRef = windowIn;
  }//end constructor

  public void windowClosed(WindowEvent e){
    System.out.println("WProc1 windowClosed test
msg");
  }//end windowClosed()

  public void windowIconified(WindowEvent e){
    System.out.println("WProc1 windowIconified
test msg");
  }//end windowIconified()

  public void windowOpened(WindowEvent e){
    System.out.println("WProc1 windowOpened test
msg");
  }//end windowOpened()

  public void windowClosing(WindowEvent e){
    System.out.println("WProc1 windowClosing test
msg");
    displayWindowRef.dispose();//generate
WindowClosed
  }//end windowClosing()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
                      "WProc1 windowDeiconified
```

```
test msg");
  }//end windowDeiconified()

  public void windowActivated(WindowEvent e){
    System.out.println("WProc1 windowActivated
test msg");
  }//end windowActivated()

  public void windowDeactivated(WindowEvent e){
    System.out.println(
                      "WProc1 windowDeactivated
test msg");
  }//end windowDeactivated()
}//end class WProc1
//===================================================
=======//

//This and the previous class can be used to
instantiate
// Listener objects. Note that this class extends
an
// Adapter class that can be used to avoid the
// requirement to define all of the methods of the
// actual Listener class named WindowListener.
This class
// overrides only two of the methods declared in
the
// interface.  It displays a message whenever one
of the
// methods is called.
class WProc2 extends WindowAdapter{

  public void windowIconified(WindowEvent e){
    System.out.println(
              "******** WProc2 windowIconified
test msg");
  }//end windowIconified()
```

```
  public void windowDeiconified(WindowEvent e){
    System.out.println(
            "******** WProc2 windowDeiconified
test msg");
  }//end windowDeiconified()

}//end class WProc2
```

More Detailed Overview of the Delegation Model

JDK 1.1 encapsulates events in a class hierarchy with the root class named **java.util.EventObject** .

The propagation of an event from a **Source** object to a **Listener** object involves calling a method on the **Listener** object and passing an object that contains encapsulated information about the event. Note that each event class may include more than one actual type of event *(as determined by the methods declared in the listener interface)* .

An event listener

A **Listener** object is an instance of a class that *implements* a specific **EventListener** interface extended from the generic **java.util.EventListener** .

An **EventListener** interface declares one or more methods that must be defined in the **Listener** class. Those methods are called by the event source in response to each specific event type handled by the interface.

The calling of these methods is the mechanism by which the **Source** notifies the **Listener** of the occurrence of an event of a specific type.

An event source

An **Event Source** is an object that "originates" or "fires" events by calling the methods of one or more **Listener** objects. The **Source** maintains a list

containing a reference to all of the **Listener** objects that have *registered* to be notified of events of that class.

Registration of event listener objects

The programmer causes **Listener** objects to be added to this list using **add *(EventType)* Listener** method calls.

Placing references to **Listener** objects on the list is often referred to as *registering* specific **Listeners** to receive notification of specific events.

Notifying listener objects

Once the list is populated *(**Listener** objects are registered)* , the **Source** object uses that list to notify each **Listener** of the occurrence of an event of the specified type without further effort on the part of the programmer.

Event sources and listeners

The **Event Source** is often a **GUI** component and the **Listener** is commonly an object of a class that implements the appropriate listener interface, but this is not a requirement. For example we will learn later how to cause a program to generate events without any physical involvement on the part of a user and a **GUI** component.

The **Listener** object could also be another AWT component that implements one or more **Listener** interfaces for the purpose of hooking **GUI** objects up to each other.

Event Hierarchy

As mentioned earlier, each specific event type is a member of a class of event types and these classes form a hierarchy of event classes.

Since a single event class may be used to represent more than one event type *(i.e. **MouseEvent** represents mouse up, mouse down, mouse drag,*

mouse move, etc.) , some event classes may also contain an "id" *(unique within that class)* that maps to its specific event types.

Setter and getter methods

There are no public fields in the event classes. Rather the data in the event is encapsulated and available only through the use of appropriate *setter* and *getter* methods.

The *setter* methods only exist for attributes on an event that can be modified by a listener. If you continue with your studies in Java, you will learn that the *setter* and *getter* methods match a *design pattern* for Java Beans.

A concrete set of event classes is defined by the AWT. In addition, programmers may define their own event types by subclassing either **java.util.EventObject** or one of the AWT event classes.

Low-level vs. Semantic Events

The AWT provides two conceptual types of events:

- low-level events
- semantic events

Low-level events

A *low-level* event is one that represents a low-level input or window-system occurrence on a visual component on the screen. As of February 1997, JDK 1.1 defined the following low-level event classes:

Note: Low-level event classes

- java.util.EventObject
    - java.awt.AWTEvent

        - java.awt.event.ComponentEvent (component resized, moved,etc.)
            - java.awt.event.FocusEvent (component got focus, lost focus)
            - java.awt.event.InputEvent
                - java.awt.event.KeyEvent (component got key-press, key-release,etc.)
                - java.awt.event.MouseEvent (component got mouse-down, mouse-move,etc.)
            - java.awt.event.ContainerEvent
            - java.awt.event.WindowEvent

As indicated earlier, some of the event classes encompass several different event types. Generally, there are corresponding **Listener** interfaces for each of the event classes, and corresponding interface methods for each of the different event types in each event class.

Semantic events

Semantic events are defined at a higher-level to encapsulate the semantics of a user interface component's model. As of February 1997, the semantic event classes defined by the JDK 1.1 version of the AWT were as follows:

Note: Semantic event classes

- java.util.EventObject
    - java.awt.AWTEvent
        - java.awt.event.ActionEvent ("do a command")
        - java.awt.event.AdjustmentEvent ("value was adjusted")

- java.awt.event.ItemEvent ("item state has changed")
- java.awt.event.TextEvent ("the value of the text object changed")

A more general event type

The *semantic* events are not tied to specific screen-based component classes, but may apply across a set of components that implement a similar semantic model. For example, a **Button** object will fire an *"action"* event when it is pressed and a **List** object will fire an *"action"* event when an item is double-clicked.

Not tied to user actions

Even though the above discussion seems to tie these event classes to user actions on screen components *(because that is the norm)* , you need to remember that there is nothing to prevent you from having your code generate events of these types completely independent of such user actions. For example, you can easily cause an **ActionEvent** to be generated and attributed to some component whenever a software timer expires. We will learn how to do this sort of thing in subsequent modules.

Event Listeners

An **EventListener** interface will typically have a separate method for each distinct event type that the event class represents. For example, the **FocusEventListener** interface defines two methods, **focusGained** and **focusLost** , one for each event type that the **FocusEvent** class represents.

Low-level listener interfaces

As of February 1997, the low-level listener interfaces defined by the JDK 1.1 version of the AWT were as follows:

Note: Low-level listener interfaces

- java.util.EventListener
    - java.awt.event.ComponentListener
    - java.awt.event.ContainerListener
    - java.awt.event.FocusListener
    - java.awt.event.KeyListener
    - java.awt.event.MouseListener
    - java.awt.event.MouseMotionListener
    - java.awt.event.WindowListener

If you match this up with the previous list of *low-level* event classes, you will see that there is a listener interface defined for each of the "leaf" classes in the hierarchy of event classes. *(In fact, there are two different listener interfaces defined for the **MouseEvent** class. This will be discussed further at the appropriate point in time.)*

Semantic listener interfaces

As of February 1997, the semantic listener interfaces defined by the AWT were as follows:

Note: Semantic event listener interfaces

- java.util.EventListener
    - java.awt.event.ActionListener
    - java.awt.event.AdjustmentListener
    - java.awt.event.ItemListener
    - java.awt.event.TextListener

There is a one-to-one correspondence between *semantic* listener interfaces and *semantic* event classes.

Event Sources

All AWT event sources support a *multicast* model for listeners. This means that multiple listeners can be added and removed from a single source. In other words, notification of the occurrence of the same event can be sent to one or more listener objects.

According to JDK 1.1 documentation,

"The API makes no guarantees about the order in which the events are delivered to a set of registered listeners for a given event on a given source. Additionally, any event which allows its properties to be modified (via setXXX() methods) will be explicitly copied such that each listener receives a replica of the original event. If the order in which events are delivered to listeners is a factor for your program, you should chain the listeners off a single listener which is registered on the source (the fact that the event data is encapsulated in a single object makes propagating the event extremely simple)."

Low-level event sources

As before, a distinction is drawn between *low-level* and *semantic* events. The source for *low-level* events will often be one of the visual component classes *(**Button** , **Scrollbar** , etc.)* because the event is bound to the actual component on the screen *(but counterfeit events can be generated)* .

As of February 1997, JDK 1.1 defined low-level listeners on the following components.

Note: Low-level listener components

- java.awt.Component

    - addComponentListener(ComponentListener l)
    - addFocusListener(FocusListener l)
    - addKeyListener(KeyListener l)
    - addMouseListener(MouseListener l)
    - addMouseMotionListener(MouseMotionListener l)
- java.awt.Container
    - addContainerListener(ContainerListener l)
- java.awt.Dialog
    - addWindowListener(WindowListener l)
- java.awt.Frame
    - addWindowListener(WindowListener l)

Must take inheritance into account

To determine all of the specific event types that can be communicated from a source object to a listener object, you must take inheritance into account.

For example, as you will see in a sample program later in this module, a source object can detect **mouse events** on a **Frame** object and notify a **MouseListener** object of the occurrence of those events even though the above list does not show a **MouseListener** on a **Frame** . This is possible because a **Frame** object indirectly extends the **Component** class, and **MouseListener** is defined for the **Component** class.

Semantic event sources

As of February 1997, JDK 1.1 defined the following semantic listeners for AWT components:

Note: Semantic listener components

- java.awt.Button
  - *addActionListener(ActionListener l)*
- java.awt.Choice (implements java.awt.ItemSelectable)
  - addItemListener(ItemListener l)
- java.awt.Checkbox (implements java.awt.ItemSelectable)
  - addItemListener(ItemListener l)
- java.awt.CheckboxMenuItem (implements java.awt.ItemSelectable)
  - addItemListener(ItemListener l)
- java.awt.List (implements java.awt.ItemSelectable)
  - *addActionListener(ActionListener l)*
  - addItemListener(ItemListener l)
- java.awt.MenuItem
  - *addActionListener(ActionListener l)*
- java.awt.Scrollbar (implements java.awt.Adjustable)
  - addAdjustmentListener(AdjustmentListener l)
- java.awt.TextArea
  - addTextListener(TextListener l)
- java.awt.TextField
  - *addActionListener(ActionListener l)*
  - addTextListener(TextListener l)

The nature of semantic events

The nature of *semantic* event types can be inferred by noticing that in some cases, different types of components support the same type of *semantic* event. For example, four different types of components are identified in the above list that can register and service *action events* .

A single **ActionEvent** listener object could be registered to be notified of *action events* on one or more components of all of these types. Of course, it would then be necessary for the code in that event handler to determine which source was responsible for generating the event if such identification is important.

An important difference between *low level* events and *semantic* events has to do with where the classes that define those events plug into the class hierarchy. *Low level* events plug in further down the class hierarchy and inherit more methods that can be used to process the event object than is the case with *semantic* events.

Adapters

Many **EventListener** interfaces are designed to listen to multiple event classes. For example, the **MouseListener** listens to *mouse-down*, *mouse-up*, *mouse-enter* , etc. The **MouseListener** interface declares a method for each of these subtypes.

When you implement an interface, you are required to define all of the methods that are declared in the interface, even if you define them with empty methods. In some cases, the requirement to define all the methods declared in an interface can be burdensome.

For this reason (and possibly for some other reasons as well), the AWT provides a set of abstract *adapter* classes that match up with the defined

interfaces. Each adapter class implements one interface and defines all of the methods declared by that interface as empty methods, thus satisfying the requirement to define all of the methods.

You can then define your listener classes by *extending* the adapter classes instead of *implementing* the listener interfaces.. This allows you the freedom to override only those methods of the interface that interest you.

Again, recall that the methods declared within an interface correspond to the individual event types contained within a corresponding event class, and the **Source** object notifies your **Listener** object of the occurrence of an event of a specific type by calling your interface method.

As of February 1997, the Adapter classes provided by the JDK 1.1 version of the AWT were as follows:

Note: Adapter classes

- java.awt.event.ComponentAdapter
- java.awt.event.FocusAdapter
- java.awt.event.KeyAdapter
- java.awt.event.MouseAdapter
- java.awt.event.MouseMotionAdapter
- java.awt.event.WindowAdapter

Filtering for Performance

Since listeners are registered to handle specific event types, they are notified only of the occurrence of those event types and are not required to deal with event types for which they are not registered. That was not the case in JDK 1.0.2 where all events passed through a common set of event handler code whether those events were of interest or not.

This filtering of events should improve performance, especially with high frequency events, such as mouse-moves.

According to the JDK 1.1 documentation,

"All platforms should see some performance improvement from reduced event traffic, but the Solaris implementation should gain exceptional improvement since it's a network-based window system."

Another Sample Program

The previous program did not terminate when the user closed the window. The following program does terminate when the user closes the window by executing a **System.exit(0)** statement in the *closing* event handler..

The previous program implemented an **Event Source** object that notified two different **Listener** objects of the occurrence of an event in the **Window** class.

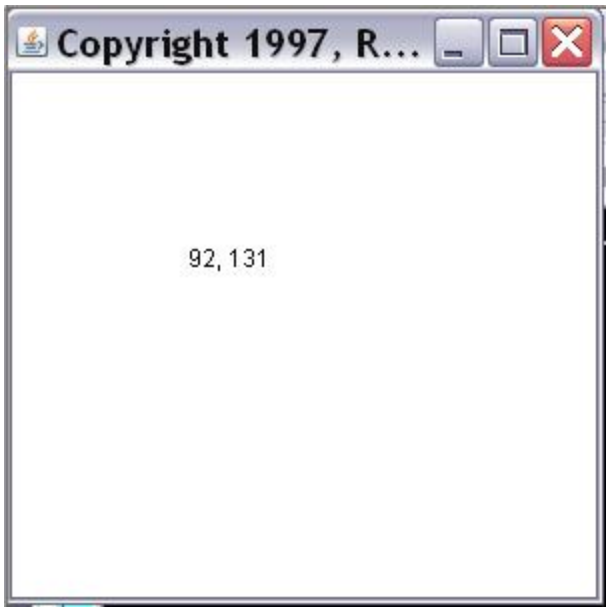
The following program implements an **Event Source** object that notifies one **Listener** object of the occurrence of an event in the **Window** class and notifies another **Listener** object of the occurrence of an event in the **Mouse** class.

This program implements a **MouseListener** interface on a **Frame** object, which is possible because the **Frame** class indirectly extends the **Component** class, and **addMouseListener** is defined on the **Component** class.

Program output

If you compile and execute this program, whenever you click the mouse inside the **Frame** , you should see the coordinates of the mouse pointer displayed above the mouse pointer as shown in Figure 2 below.

Figure 2 - Screen output from Event09.



Event09 Interesting Code Fragments

The first interesting code fragment is the definition of the controlling class. The **main** method for this class instantiates a **GUI** object where all the real work is done. Listing 10 shows the class named **Event09** including the **main** method.

Listing 10 - Beginning of the class named Event09.

```
public class Event09 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class Event09
```

The paint and repaint methods

Various tutorial lessons on my website discuss the relationship between the **paint** method and the **repaint** method. If you are unfamiliar with that relationship, you may need to go back and review it.

In this program, we are going to override the **paint** method to make it possible for us to draw coordinate information on the screen.

In order to override the **paint** method, we need to extend the **Frame** class. By doing so, we can define our own version of the **Frame** class where we have the ability to override the **paint** method. Listing 11 *extends* **Frame** into **MyFrame** and overrides the **paint** method.

Listing 11 - The class named MyFrame.

```
class MyFrame extends Frame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
              "" + clickX + ", " + clickY,
clickX, clickY);
  }//end paint()
}//end class MyFrame
```

Storing and using coordinate information

For future reference, note that this class contains a couple of instance variables that will be used to store X and Y coordinate information.

The overridden version of the **paint** method causes the coordinate values stored in **clickX** and **clickY** to be displayed in the client area of the **Frame** object at the location specified by the values of those two instance variables.

The drawString method

The **drawString** method requires three parameters. The first parameter is the string to draw on the screen. The next two parameters are the coordinate values in pixels where the string is to be drawn.

As is normally the case, coordinate values are specified relative to the upper left corner of the object being drawn on. The above overridden version of the **drawString** method converts the coordinate values to a **String** , and draws that string at the location specified by the coordinate values.

The class named GUI

As before, the **GUI** class is rather long. Therefore, I will break it up and discuss it in fragments.

The first fragment that I will discuss is the constructor that begins in Listing 12 . The important thing to note here is that we are not instantiating an object of type **Frame** . Instead, we are instantiating an object of type **MyFrame** that is our extended version of **Frame** .

Other than that, you should be familiar with the code in Listing 12 by now.

Listing 12 - Beginning of the class named GUI.

Listing 12 - Beginning of the class named GUI.

```
class GUI {
  public GUI(){//constructor
    //Create a new Frame object, set size,
title, etc.
    MyFrame displayWindow = new MyFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Copyright 1997,
R.G.Baldwin");
    displayWindow.setVisible(true);
```

Register two listener objects

Listing 13 uses standard syntax to instantiate and register two different **Listener** objects on the **MyFrame** object. The first is a **WindowListener** object that will terminate the program when the user closes the **MyFrame** object.

The second is a **MouseListener** object that will process *mouse events* on the **MyFrame** object.

It is typical to register listener objects as *anonymous objects* in those cases where a specific reference to the listener object is not otherwise needed. *(Don't confuse anonymous objects with anonymous classes, which is the topic of a future module.)*

Listing 13 - Register two listener objects.

Listing 13 - Register two listener objects.

```
    displayWindow.addWindowListener(new
WProc1());
    displayWindow.addMouseListener(
                                new
MouseProc(displayWindow));
  }//end constructor
}//end class GUI definition
```

Listing 13 ends the definition of the **GUI** class. That brings us to the definition of **Listener** classes for the **WindowListener** and the **MouseListener** interfaces.

The MouseListener class

I will begin with the **MouseListener** class named MouseProc . This class *extends* the **MouseAdapter** class and does not directly *implement* the *MouseListener* interface. This saves us the trouble of having to create empty methods for event types that we are not interested in.

Objects of this class that are registered on the **Source** will be notified whenever a **mousePressed()** event occurs on the **Source.** This will cause the **mousePressed** method to be called. The **mousePressed** method will display the coordinates of the mouse pointer when the mouse is pressed on the source object.

The mousePressed method

When the **mousePressed** method is called, it receives a reference to an object of type **MouseEvent** as a parameter. Different types of events encapsulate different types of information in the object that is passed as a parameter to the event handler method. This particular object contains a variety of information, including the coordinates of the mouse pointer when the event occurred.

Extract and save mouse pointer coordinates

The code in the overridden **mousePressed()** method extracts that coordinate information from the object and stores it in the instance variables named **clickX** and **clickY** of the **Source** object.

Then it calls the **repaint** method on the source object, causing the source object to be repainted on the screen. This in turn causes the overridden **paint** method discussed earlier to be called, which displays the new coordinate information on the screen in the proper location.

The class named MouseProc

Listing 14 contain the definition of the MouseProc class, including the overridden **mousePressed** method.

Listing 14 - The class named MouseProc.

Listing 14 - The class named MouseProc.

```
class MouseProc extends MouseAdapter{
  MyFrame refToWin; //save a reference to the
source here

  MouseProc(MyFrame inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  public void mousePressed(MouseEvent e){
    refToWin.clickX = e.getX();
    refToWin.clickY = e.getY();

    refToWin.repaint();
  }//end mousePressed()
}//end class MouseProc
```

The class named WProc1

Finally, we come to the class that is used to instantiate a listener object that terminates the program when the user closes the **MyFrame** object *(see Listing 15)* . You will be seeing this class over and over as you review the sample programs in upcoming modules.

Listing 15 - The class named WProc1.

Listing 15 - The class named WProc1.

```
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
```

The **WProc1** class extends the **WindowAdapter** class and overrides the method named **windowClosing** .

The **windowClosing** method is called when the user clicks the red X in the upper right corner of Figure 2 .That causes the static **exit** method of the **System** class to be called, which in turn causes the program to terminate.

Event09 Program Listing

A complete listing of the program named **Event09** is provided in Listing 16 .

Listing 16 - Complete listing of Event09.

```
/*File Event09.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled under JDK
1.1

Illustrates the use of Event Sources, Event
Listeners, and
Adapters in the Delegation Event Model.

This program instantiates a Listener object to
process
mouse events. When a mouse press occurs in a Frame
```

```
object,
the program gets the coordinates and then displays
those
coordinates near the point of the mouse press.

This program was tested using JDK 1.1.3 under
Win95.
**********************************************
********/
import java.awt.*;
import java.awt.event.*;

public class Event09 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class Event09
//=============================================
=======//

//Subclass Frame in order to override the paint
method.
class MyFrame extends Frame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
              "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paint()
}//end class MyFrame
//=============================================
=======//

//The following class is used to instantiate a
// graphical user interface object.
```

```
class GUI {
  public GUI(){//constructor
    //Create a new Frame object, set size, title,
etc.
    MyFrame displayWindow = new MyFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Copyright 1997,
R.G.Baldwin");
    displayWindow.setVisible(true);

    //Instantiate and register an anonymous
Listener
    // object that will terminate the program when
the
    // user closes the Frame.
    displayWindow.addWindowListener(new WProc1());

    //Instantiate and register an anonymous
Listener
    // object that will process mouse events to
determine
    // and display the coordinates when the user
presses
    // the mouse button in the client area of the
Frame.
    displayWindow.addMouseListener(
                                new
MouseProc(displayWindow));
  }//end constructor
}//end class GUI definition
//=============================================
=======//

//This listener class monitors for mouse presses
and
// displays the coordinates of the mouse pointer
when the
```

```
// mouse is pressed on the source object. Note
that this
// class extends is an adapter class.
class MouseProc extends MouseAdapter{
  MyFrame refToWin; //save a reference to the
source here

  MouseProc(MyFrame inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  //Override the mousePressed method to determine
and
  // display the coordinates when the mouse is
pressed.
  public void mousePressed(MouseEvent e){
    //Get X and Y coordinates of mouse pointer and
store
    // in an instance variable of the Frame object
    refToWin.clickX = e.getX();
    refToWin.clickY = e.getY();

    //Force the Frame object to be repainted in
order to
    // display the coordinate information.
    refToWin.repaint();
  }//end mousePressed()
}//end class MouseProc
//=========================================
=======//

//The following listener is used to terminate the
program
// when the user closes the frame.  Note that this
class
// extends an adapter class.
class WProc1 extends WindowAdapter{
```

```
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===========================================
=======//
```

Summary

In the simplest case, you can handle events as defined in JDK 1.1 with the following steps.

Step 1

Define a **Listener** class for a specific class of events by either *implementing* the listener interface that matches that class of events, or *extending* the corresponding adapter class.

Step 2

Define or override the interface methods in the **Listener** class, for each specific event type in the event class, to implement the desired behavior of the program upon occurrence of an event.

If you *implement* the listener interface, you must define all interface methods. If instead you *extend* the corresponding adapter class, you can override only those methods that tie to event types of interest.

Step 3

Write code that instantiates objects of the **Source** class and the **Listener** class and *registers* the listener object on the source object for notification of

events generated by the source object.

You can use code such as the following for registration:

displayWindow. **addMouseListener** (mouseProcCmd);

In this code fragment,

- **displayWindow** is a reference to the object that fires the event,
- **mouseProcCmd** is the name of a reference to the listener object, and
- **addMouseListener** is the method that registers the listener object to receive mouse events from the object referred to by **displayWindow** .

This statement will cause the object named **mouseProcCmd** to be notified of all events generated by **displayWindow** which are part of the class of *mouse* events.

The notification takes the form of calling a method in the **mouseProcCmd** object where there must be a corresponding method for each specific event type in the class of *mouse* events. *(Some of those methods can be empty shells if you have no interest in some of the event types).*

Comments

As mentioned earlier, this is the procedure for the simplest cases. It is possible to make the situation more complicated. For example, a single **Source** object can be required to notify two or more different **Listener** objects of the occurrence of an event of the same class on a single screen object.

The **Source** object can also be required to notify two or more different **Listener** objects of the occurrence of events of two different classes on a single screen object.

A single **Listener** object can be registered to receive notification of the occurrence of events of a given class on more than one source object. In that case, it is normally necessary for the code in the **Listener** object to

crack open the event object to determine which screen object was responsible for the event *(if it matters)* .

Also, the JDK 1.1 documentation indicates that it is possible to have event-style communication between objects that are not screen objects. Subsequent modules will investigate a number of these possibilities.

Review

The original HTML version of this module contains a number of review questions with answers. If interested, you can take a look at those review questions .

Run the program

I encourage you to copy the code from Listing 9 and Listing 16 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Event Handling in JDK 1.1, A First Look, Delegation Event Model
- File: Java0080.htm
- Published: 11/17/13
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java0081 Java OOP Swing and the Delegation Event Model
A discussion of the use of the delegation event model with Swing.

Table of Contents

- Preface
    - Viewing tip
        - Listings
- Introduction
- Sample Program
    - Interesting Code Fragments
    - Program Listing
- Another Sample Program
    - Interesting Code Fragments
    - Program Listing
- Summary
- Run the programs
- Miscellaneous
- Listings

Preface

This module is one in a collection of modules designed for teaching *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.

(Editor's note: As you read this module, you will see that it was originally written around 1997. However, despite many improvements in Java since then, most of what was true then is still true in 2013.)

This module makes several references to my website, which is located at http://www.dickbaldwin.com/toc.htm . A copy of the original html version of this module is available here .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

- Listing 1 . Beginning of the constructor for the GUI class.
- Listing 2 . Complete listing of the program named SwingEvent08.
- Listing 3 . The class named MyFrame.
- Listing 4 . Complete listing of the program named SwingEvent09.

Introduction

To begin with, what is Swing?

Swing is the name given to a new set of lightweight components developed by Sun to supplement *(and possibly replace)* the components in the AWT. *(The general topic of lightweight components is covered in a separate module.)* With the exception of top-level containers, Swing components are developed completely using Java and don't depend on the *peer* component rendering provided by the operating system.

Swing components have several advantages over AWT components, and probably some disadvantages as well.

One of the advantages is that because the components are not rendered on the screen by the operating system, the look and feel of a component does

not change as the application or applet is executed on different platforms running under different operating systems.

Furthermore, it is possible to cause Swing components to mimic the look and feel of a specific platform no matter what platform the program is running on. This is known as *pluggable look and feel* . *(A complete module is dedicated to pluggable look and feel.)*

In any event, Swing components are here to stay, and from this point forward in these tutorial modules, we will attempt to incorporate Swing into the modules in a meaningful way.

Swing components support the JDK 1.1 Delegation Event Model . They do not support the JDK 1.0 event model, so if you want to use Swing components, you must program using the Delegation Event Model.

In some cases, conversion of an application or applet from AWT components to Swing components involves nothing more complex than replacing the AWT components with corresponding Swing components. For example, the Swing component that corresponds to the AWT **Button** component is the Swing **JButton** component.

In all cases that you use Swing you will need to import, or otherwise recognize the package that contains the Swing classes so that the compiler and interpreter will have access to those classes. Note that the way to do this will probably change between JDK 1.1.x and JDK 1.2, but hopefully the change won't be too great. *(Rumor has it that Swing will be more tightly integrated into JDK 1.2 than is the case with JDK 1.1.)*

This module is designed as a follow-on to the earlier module titled " Event Handling in JDK 1.1, A First Look, Delegation Event Model ." As such, material presented in that module will not be repeated here.

The two AWT sample programs in that module were converted to Swing programs for presentation in this module using the simple substitution procedure described above. As you will see, the Swing version of one of the sample program behaves essentially the same as its AWT counterpart. The

Swing version of the other sample program does not behave the same as its AWT counterpart.

So, where does that leave us? Well, fortunately, from an event handling viewpoint, Swing components operate the same as AWT components *(except that Swing provides a number of new event types)* . From other viewpoints, Swing components may or may not behave the same as their AWT counterparts.

In addition, many Swing components don't have an AWT counterpart. A number of new and exciting components are included in the Swing library that don't exist in the AWT *(tooltips, progress bars, trees, etc.)* .

Since this module is primarily concerned with event handling, we will pursue the Swing components from an event-handling viewpoint in this module and leave other considerations surrounding Swing components for subsequent modules. As you will see, this module will raise some interesting questions regarding the behavior of Swing components, but won't provide the answers to those questions.

Sample Program

The sample program in this section, named SwingEvent08 , was created by replacing each instance of **Frame** in the program named **Event08** , with an instance of **JFrame** . In addition, an import declaration was added to cause the Swing classes to be accessible to the compiler and the interpreter.

(Note that as of JDK 1.7.x, the import declaration must read ***import javax.swing.*;*** *instead of* ***import com.sun.java.swing.*;*** *as shown in Listing 2 .)*

The event handling in this program is exactly the same as in **Event08** from the earlier module. If you haven't reviewed that program, and the module on the Delegation Event Model, you should probably go back and review it now.

This program illustrates the use of *Event Sources* , *Event Listeners* , and *Adapters* in the *Delegation Event Model* for Swing components.

Briefly, this application instantiates an object that creates a user interface consisting of a simple **JFrame** object. This object is an *Event Source* that notifies two different *Event Listener* objects of **Window** events.

One of the **Listener** objects *implements* the **WindowListener** interface and defines all of the methods declared in that interface.

The other **Listener** object *extends* the **Adapter** class named **WindowAdapter** . As explained in the earlier module, the purpose of **Adapter** classes *(as used in this context)* is to implement the **Listener** interfaces and to define all of the methods with empty methods. *(Adapter classes are viewed in a broader context in the advanced modules on the Reflection API.)*

Classes that *extend* the **Adapter** classes can then selectively override only those methods of interest. This Listener object overrides only two of the methods.

Note that this application does not terminate and return control to the operating system. You must forcefully terminate it.

This program was tested using JDK 1.1.6 and Swing 1.0.1 under Win95. *(Also, it was recently tested using JDK 1.7 running under Windows Vista after modifying the import directive mentioned above.)*

The output produced by running the program is presented in the comments at the beginning of the program listing.

Interesting Code Fragments

Listing 1 provides the first and only code fragment, that I am going to show and discuss. This fragment shows the beginning of the constructor for the **GUI** class. The fragment illustrates that this program uses the **JFrame** class to instantiate a top-level container for the graphical user interface instead of a **Frame** object as in the earlier module .

If you compile and run this program, you will see that it behaves essentially the same as its AWT counterpart named **Event08** that was discussed extensively in the earlier module .

Program Listing

A complete listing of the program is provided in Listing 2 .

Another Sample Program

This is a Swing version of the program named **Event09** from the earlier module .

The purpose of this program is to illustrate that in some cases, conversion of an AWT program to a Swing program isn't as simple as importing the Swing package and replacing the AWT components with their Swing counterparts.

That is what was done with this program. This program is identical to the AWT program named **Event09** except that all instances of **Frame** were replaced by **JFrame** and the Swing package was imported.

The intended behavior of this program *(go back and review **Event09** if necessary)* is to display the coordinates of the mouse pointer each time the mouse is clicked inside the client area of the **JFrame** .

Initially, this program appears to work correctly. As each new click occurs on the client area of the **JFrame** , a new pair of coordinate values is displayed. However, the previous pairs of coordinate values does not disappear as is the case with the program named **Event09** . Thus, the behavior is significantly different under JDK 1.1.6 and Swing 1.0.1 under Win95. *(The behavior is similarly different under JDK 1.7x and Windows Vista.)*

Furthermore, in some cases, while this GUI is on the screen, if focus is transferred to a different application, all of the coordinate values except the

last one that was created will disappear.

It will be left as an exercise for the student to dig into the documentation on Swing in order to understand and explain this behavior.

As explained earlier, this module is concerned with event handling in Swing and this program is adequate to illustrate the use of the Delegation Event Model with Swing. We will leave the subtle behavior of the Swing components for discussion in subsequent modules.

If you compile and execute this program, whenever you click the mouse inside the **JFrame** , you should see the coordinates of the mouse pointer displayed above the mouse pointer.

Interesting Code Fragments

Listing 3 shows the only code fragment that will be highlighted for this program in this module. This fragment illustrates how this program extends **JFrame** into a new class named **MyFrame** to make it possible to override the **paint()** method of the **JFrame** class. Recall that in the program named **Event09** , the **Frame** class was extended in a similar way.

Program Listing

A complete listing of this program is provided in Listing 4 .

Summary

Swing provides a new set of lightweight components that can be programmed using the Delegation Event Model first introduced in JDK 1.1.

The Swing components cannot be programmed using the event model from JDK 1.0.

In some cases, all that is necessary to convert a program built around AWT components to a program built around Swing components is to import the Swing classes and replace all instances of AWT components with their Swing counterparts.

In other cases, a simple substitution as described above will not produce the same behavior.

Many of the Swing components do not have an AWT counterpart. The Swing classes contain a number of components that were never added to the AWT component library.

This module has dealt exclusively with the use of the Delegation Event Model for programming Swing components and has left some unanswered questions regarding how the behavior of Swing components does, and in some cases does not, mirror that of their AWT counterparts.

Run the programs

I encourage you to copy the code from Listing 2 and Listing 4 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Swing and the Delegation Event Model
- File: Java0081.htm
- Published: 11/17/13
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Listings

Listing 1 . Beginning of the constructor for the GUI class.

Listing 1 . Beginning of the constructor for the GUI class.

```
class GUI{
  public GUI(){//constructor
    //Create a new JFrame object
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");
```

Listing 2 . Complete listing of the program named SwingEvent08.

```
/*File SwingEvent08.java Copyright 1998,
R.G.Baldwin
Rev 05/09/98

This is a Swing version of the program named
Event08.

The purpose of this program is to illustrate that
in many
respects, programming with Swing components is no
different
from programming with AWT components.

Conversion of this program from AWT to Swing
involved
nothing more complex than using the search and
replace
feature of an editor to replace all instances of
Frame
with JFrame and to import the swing package.

Illustrates the use of Event Sources, Event
```

Listeners, and
Adapters in the Delegation Event Model for Swing components.

Briefly, this application instantiates an object which;
creates a user interface consisting of a simple JFrame;
object. This object is an Event Source which notifies two
different Event Listener objects of Window events.

One of the Listener objects implements the WindowListener
interface and overrides all of the methods declared in;
that interface.

The other Listener object extends the Adapter class named;
WindowAdapter. The purpose of Adapter classes is to
implement the Listener interfaces and to define all of
the methods with empty methods. Classes which extend the
Adapter classes can then selectively override only those
methods of interest. This Listener object overrides only
two of the methods.

Note that this application does not terminate and return
control to the operating system. You must forcefully
terminate it.

Tested using JDK 1.1.6 and Swing 1.0.1 under Win95.

When executed, this application places a simple empty;
JFrame object on the screen.

Starting the program produces screen output similar to
the following:

```
WProc1 windowActivated test msg
Wproc1 windowOpened test msg
```

Pressing the minimize button on the JFrame produces the;
output similar to the following:

```
WProc1 windowIconified test msg
******** WProc2 windowIconified test msg
WProc1 windowDeactivated test msg
```

Restoring the JFrame after minimization produces the;
output similar to the following:

```
WProc1 windowActivated test msg
WProc1 windowDeiconified test msg
******** WProc2 windowDeiconified test msg
WProc1 windowActivated test msg
```

Closing the JFrame by pressing the X-icon in the upper;
right of the JFrame produces output similar to the;
following:

```
WProc1 windowClosing test msg
WProc1 windowDeactivated test msg
WProc1 windowClosed test msg
**********************************************
********/

import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class SwingEvent08 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
}//end class SwingEvent08
//=============================================
=======//

//The following class is used to instantiate a
user;
// interface object, to instantiate two Listener
objects,
// and to register those two objects for
notification;
// whenever a Window event occurs.
class GUI{
  public GUI(){//constructor
    //Create a new JFrame object
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");

    //Instantiate two Listener objects which will
process
```

```
    // Window events
    WProc1 winProcCmd1 = new
WProc1(displayWindow);
    WProc2 winProcCmd2 = new WProc2();

    //Register the Listener objects for
notification of
    // Window events. This object is the Event
Source.
    displayWindow.addWindowListener(winProcCmd1);
    displayWindow.addWindowListener(winProcCmd2);

    //windowActivated and windowOpened test
messages
    // are produced here
    displayWindow.setVisible(true);

  }//end constructor
}//end class GUI definition
//=============================================
=======//

//The following two classes can be used to
instantiate;
// Listener objects. Note that this class
implements the;
// WindowListener interface.  This requires that
all the
// methods declared in the interface be overridden
in this
// class. This class overrides all of the methods
and
// displays a descriptive message whenever one of
the
// methods is invoked.
class WProc1 implements WindowListener{
  //used to save a reference to the JFrame object
```

```
  JFrame displayWindowRef;

  WProc1(JFrame windowIn){//constructor
    // save ref to JFrame object
    this.displayWindowRef = windowIn;
  }//end constructor

  public void windowClosed(WindowEvent e){
    System.out.println("WProc1 windowClosed test
msg");
  }//end windowClosed()

  public void windowIconified(WindowEvent e){
    System.out.println("WProc1 windowIconified
test msg");
  }//end windowIconified()

  public void windowOpened(WindowEvent e){
    System.out.println("WProc1 windowOpened test
msg");
  }//end windowOpened()

  public void windowClosing(WindowEvent e){
    System.out.println("WProc1 windowClosing test
msg");
    displayWindowRef.dispose();//generate
WindowClosed
  }//end windowClosing()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
                        "WProc1 windowDeiconified
test msg");
  }//end windowDeiconified()

  public void windowActivated(WindowEvent e){
    System.out.println("WProc1 windowActivated
```

```
test msg");
  }//end windowActivated()

  public void windowDeactivated(WindowEvent e){
    System.out.println(
                      "WProc1 windowDeactivated
test msg");
  }//end windowDeactivated()
}//end class WProc1
//=============================================
=======//

//This and the previous class can be used to
instantiate;
// Listener objects. Note that this class extends
an;
// Adapter class which can be used to avoid the;
// requirement to define all of the methods of the
// actual Listener class named WindowListener.
This class
// overrides only two of the methods declared in
the;
// interface.  It displays a message whenever one
of the
// methods is invoked.
class WProc2 extends WindowAdapter{

  public void windowIconified(WindowEvent e){
    System.out.println(
              "******** WProc2 windowIconified
test msg");
  }//end windowIconified()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
             "******** WProc2 windowDeiconified
test msg");
```

```
  }//end windowDeiconified()

}//end class WProc2
```

Listing 3 . The class named MyFrame.

```
class MyFrame extends JFrame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
              "" + clickX + ", " + clickY,
clickX, clickY);
  }//end paint()
}//end class MyFrame
```

Listing 4 . Complete listing of the program named SwingEvent09.

```
/*File SwingEvent09.java Copyright 1998,
R.G.Baldwin
Rev 5/9/98 by RGB
This is a Swing version of the program named
Event09.

The purpose of this program is to illustrate that
in some
cases, conversion of an AWT program to a Swing
```

program
isn't as simple as importing the Swing package and replacing
the AWT components with Swing components.

That is what was done with this program. In other words,
this program is identical to the AWT program named Event09
except that all instances of Frame were replaced by JFrame
and the Swing package was imported.

At the surface, the program appears to work correctly.
However, as each new click occurs on the client area of
the JFrame, a new pair of coordinate values is displayed,
but the previous pairs of coordinate values don't disappear
as is the case with the program named Event09.

Furthermore, in some cases, while this GUI is on the screen,
if focus is transferred to a different application, all of
the coordinate values except the last one created will;
disappear.

It will be left as an exercise for the student to dig into
the (currently very sparse) documentation on Swing in order
to understand and explain this behavior.

```
Illustrates the use of Event Sources, Event
Listeners, and;
Adapters in the Delegation Event Model for Swing.

This program instantiates a Listener object to
process;
mouse events. When a mouse press occurs in a
JFrame object,
the program gets the coordinates and then displays
those
coordinates near the point of the mouse press.

Tested using JDK 1.1.6 and Swing 1.0.1 under
Win95.
**********************************************
********/
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;

public class SwingEvent09 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class SwingEvent09
//=============================================
=======//

//Subclass JFrame in order to override the paint
method.
class MyFrame extends JFrame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
              "" + clickX + ", " + clickY, clickX,
```

```
clickY);
  }//end paint()
}//end class MyFrame
//=============================================
=======//

//The following class is used to instantiate a;
// graphical user interface object.
class GUI {
  public GUI(){//constructor
    //Create a new JFrame object, set size, title,
etc.
    MyFrame displayWindow = new MyFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");
    displayWindow.setVisible(true);

    //Instantiate and register an anonymous
Listener;
    // object which will terminate the program
when the;
    // user closes the JFrame.
    displayWindow.addWindowListener(new WProc1());

    //Instantiate and register an anonymous
Listener;
    // object that will process mouse events to
determine
    // and display the coordinates when the user
presses
    // the mouse button in the client area of the
JFrame.
    displayWindow.addMouseListener(
                                new
MouseProc(displayWindow));
  }//end constructor
```

```
}//end class GUI definition
//===========================================
=======//

//This listener class monitors for mouse presses
and;
// displays the coordinates of the mouse pointer
when the
// mouse is pressed on the source object. Note
that this
// class extends is an adapter class.
class MouseProc extends MouseAdapter{
  MyFrame refToWin; //save a reference to the
source here

  MouseProc(MyFrame inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  //Override the mousePressed method to determine
and;
  // display the coordinates when the mouse is
pressed.
  public void mousePressed(MouseEvent e){
    //Get X and Y coordinates of mouse pointer and
store
    // in an instance variable of the JFrame
object
    refToWin.clickX = e.getX();
    refToWin.clickY = e.getY();

    //Force the JFrame object to be repainted in
order to
    // display the coordinate information.
    refToWin.repaint();
  }//end mousePressed()
}//end class MouseProc
```

```
//=======================================================//

//The following listener is used to terminate the program
// when the user closes the frame.  Note that this class
// extends an adapter class.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//=======================================================//
```

-end-

Java082 Sharing a Listener Object Among Visual Components
Previous lessons discussed the sharing of a single event Source object among two or more Listener objects for low-level events. This lesson discusses the sharing of a single Listener object among two or more visual components for low-level events generated by any of the visual components.

Revised: Thu Jul 07 15:48:00 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- First sample program
    - Interesting code fragments
        - The Event Source Objects
        - Unique names
        - Instantiate and register mouse listener object
        - Instantiate and register window listener object
        - The MouseListener object
        - Identify the event source
        - Cracking the MouseEvent object
        - The WindowListener Object

    - Complete program listing for Event10

- Second sample program

    - Changes relative to Event10
    - Complete program listing for Event11

- Review

    - Question 1
    - Question 2
    - Question 3
    - Question 4

- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

Students in Prof. Baldwin's **ITSE 2327 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Screen output from Event10.

- Figure 2. Screen output from Event11.
- Figure 3. Screen output for Question 1.
- Figure 4. Screen output for Question 2.

Listings

- Listing 1. An event source object.
- Listing 2. Give the source objects unique names.
- Listing 3. Instantiate and register mouse listener object.
- Listing 4. Instantiate and register window listener object.
- Listing 5. Identify the event source.
- Listing 6. The program named Event10.
- Listing 7. The program named Event11.
- Listing 8. Solution to Question 1.
- Listing 9. Solution to Question 2.
- Listing 10. Solution to Question 3.
- Listing 11. Solution to Question 4.

Introduction

Earlier lessons discussed the sharing of a single event *source* object among two or more *listener* objects for *low-level* events.

This lesson discusses the sharing of a single listener object among two or more visual components for *low-level events* generated by any of the visual components.

The event-handling method of the listener object

- responds to each event,
- determines which visual component generated the event, and
- takes the appropriate action.

The primary issue in this type of operation boils down to determining which component generated the event.

This can be accomplished by assigning a unique name to each visual component when it is instantiated and using that name to distinguish among the visual components when an event occurs.

This approach is completely straightforward and intuitive. An improved approach to accomplishing the same objective is illustrated in some sample programs near the end of the lesson. While possibly less intuitive, the improved approach is more object-oriented and probably easier to maintain.

First sample program

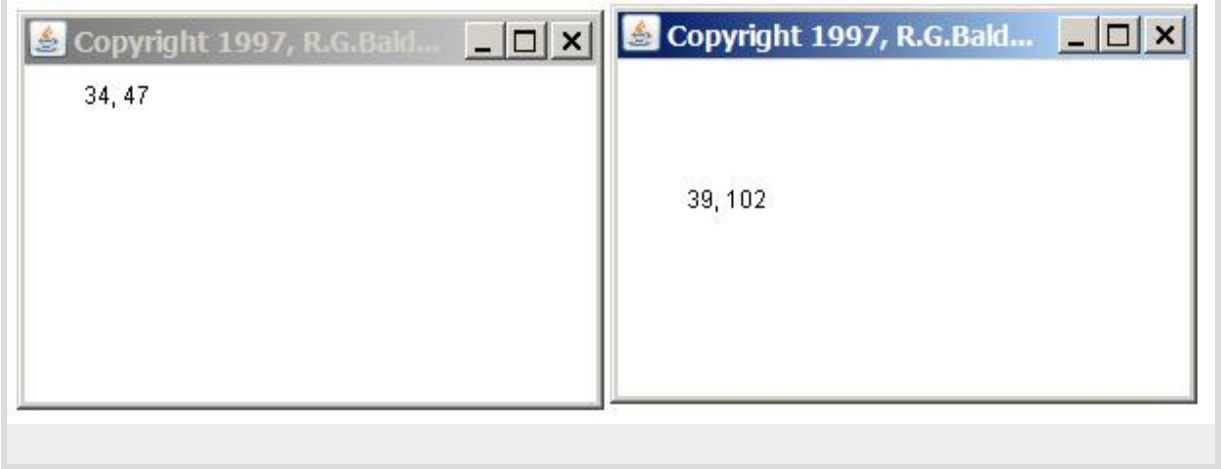
This program illustrates the sharing of a single listener object among two different visual components of the same type.

The program detects mouse events occurring on either of two different **Frame** objects. It distinguishes between the two objects on the basis of the component name, and displays the coordinates of a mouse click on whichever object experienced the mouse click.

This program was originally tested using JDK 1.1.3 under Win95. More recently it was tested using Java 8 and Windows 7.

Note that when the program starts, the two visual components overlay one another. It is necessary to physically move one of them in order to access the other as shown in Figure 1 .

Note: Figure 1. Screen output from Event10.



Interesting code fragments

This program is really quite simple. The only new code involves cracking the event object to obtain the name of the visual component that generated the **mousePressed** event.

The **main** method instantiates an object of type **GUI** *(named gui)* which serves the purpose of providing the visual interface.

The **Frame** class is *extended* into a new class named **MyFrame** to make it possible to override the **paint** method of the class. This is necessary to display the coordinates of mouse clicks on the interior of the frame using the graphics method named **drawString** .

The Event Source Objects

The constructor of the **GUI** class instantiates two objects of type **MyFrame** and makes them visible. This is accomplished using code such as that shown in Listing 1 .

Note: Listing 1. An event source object.

```
    MyFrame myFrame1 = new MyFrame("Frame1");
    myFrame1.setVisible(true);
```

The reference variables for the two objects are named **myFrame1** and **myFrame2** .

Unique names

When they are instantiated, unique names *(**Frame1** and **Frame2**)* are given to the objects using code in the constructor *(see Listing 2)* for the extended **Frame** objects *(the **MyFrame** class extends the **Frame** class in order to override its **paint** method).*

Note: Listing 2. Give the source objects unique names.

```
  MyFrame(String name){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setSize(300,200);
    //Name used to distinguish between the two
objects
    setName(name);
  }//end constructor
```

The unique names are used later by the listener object to determine which object generated a mouse event.

Instantiate and register mouse listener object

The constructor in the **GUI** class also uses the code in Listing 3 to instantiate a single listener object which will process low-level mouse events on either of the two visual objects.

Note: Listing 3. Instantiate and register mouse listener object.

```
    MouseProc mouseProcCmd = new
MouseProc(myFrame1,myFrame2);
    myFrame1.addMouseListener(mouseProcCmd);
    myFrame2.addMouseListener(mouseProcCmd);
```

The first statement in Listing 3 instantiates the new listener object named **mouseProcCmd** passing references to the two visual components as parameters.

The next two statements add that listener object (*register* the listener object) to a list of listener objects that are automatically notified whenever mouse events occur on the visual objects referred to as **myFrame1** and **myFrame2** , respectively.

You will recall that once the programmer causes the name of a listener object to be added to the list, no further programming effort is required to cause the notification to occur.

The notifications are carried out by invoking specific overridden instance methods of the listener object upon the occurrence of a specific types of mouse events.

The declarations for all of the methods that match up with all of the possible mouse event types are defined by the **MouseListener** interface that matches the **MouseEvent** class.

The class from which the listener object is instantiated must define, either directly or indirectly, all the methods declared in the **MouseListener** interface.

Instantiate and register window listener object

In addition to the *registration* of the **MouseListener** object to receive mouse events, the program also instantiates and *registers* a listener object that monitors for **Window** events and terminates the program whenever the user closes either of the two visual objects. In this case, the code in the listener object makes no attempt to distinguish between the two visual objects. The instantiation and registration code is shown in Listing 4 .

Note: Listing 4. Instantiate and register window listener object.

```
WProc1 winProcCmd1 = new WProc1();
myFrame1.addWindowListener(winProcCmd1);
myFrame2.addWindowListener(winProcCmd1);
```

The MouseListener object

Most of the programming complexity is tied up in the **mouseListener** object, and even that isn't very complicated.

The central issue for the code in the listener object is how to determine which one of several visual components generated an event.

This particular listener object only responds to **mousePressed** events, but the following information applies to all of the different types of mouse events, and probably to most or all of the low-level events as well.

The **MouseProc** *(listener)* class in this program

- extends the **MouseAdapter** class and
- overrides the **mousePressed** method that is declared in the **MouseListener** interface.

You will recall that the **MouseAdapter** class overrides all of the methods of the **MouseListener** interface with empty methods, thus freeing our code from the requirement to override all of those methods.

When the **mousePressed** method is invoked, an object of type **MouseEvent** *(known locally as **e**)* is passed in as a parameter.

Identify the event source

The statement in Listing 5 was used in the **MouseListener** object to determine if the name of the object that generated the event is **Frame1** .

Note: Listing 5. Identify the event source.

```
if( e.getComponent().getName().compareTo("Frame1")
== 0 ){
```

If it is determined that the name of the component that generated the event is **Frame1** , code is executed to display the coordinates of the mouse pointer on the visual object named **Frame1** .

Otherwise, an **else** clause is used to display the coordinate information on the visual object named **Frame2** . *(No provisions were made for the event to have been generated by any visual objects other than these two.)*

The code to display the coordinate information is essentially the same as was discussed in a similar program in an earlier lesson, so it won't be discussed again here.

Cracking the MouseEvent object

Now let's take a look at the details of cracking the **MouseEvent** object to obtain the name of the visual component that generated the event.

The **getComponent** method is a method of the **java.awt.event.ComponentEvent** class which, according to the JDK 1.1 documentation, *"Returns the component where this event originated."* It is returned as an object of type **Component** which is acted upon by the **getName** method in Listing 5 .

The **getName** method was added to the **java.awt.Component** class by JDK 1.1. This method *"Gets the name of the component"* and returns it as a **String** object. The **String** object is acted upon by the **compareTo** method in Listing 5 .

The **compareTo** method is a standard method of the **String** class, carried forward from JDK 1.0.2, that can be used to compare two **String** objects. It is used to determine if the name of the component matches the String " *Frame1"* .

Note that it is also possible to perform tests directly on the **MouseEvent** object to match it to a component name. The procedure for doing this will be included in a future lesson.

By the way, in case you haven't noticed, the **java.awt.event** package is different from the **java.awt** package. The **java.awt.event** package was added in JDK 1.1. It can be very confusing if you drop into the **java.awt** package of the documentation when you really need to be in **java.awt.event** .

The WindowListener Object

This program also contains a **WindowListener** object, which terminates the program whenever the user closes either of the **Frame** objects. Except for the fact that this object is registered to receive **Window** events from either of the two **Frame** objects, it is no different from similar listeners used in an earlier sample program, and therefore, won't be discussed further here.

Note that an improved version of this program is presented at the end of this lesson. The improved version does not require the establishment of source object names, and does not require the passing of references to the constructor for the **WindowListener** in order to distinguish between the two sources. You should become familiar with the methodology used in the improved version as well as the methodology used in the following version.

Complete program listing for Event10

A complete listing of the program named **Event10** is provided in Listing 6 .

Note: Listing 6. The program named Event10.

```
/*File Event10.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled and run
under
JDK 1.1

The program illustrates the sharing of a single
listener
object among two different visual components of
the same
type.

The program detects mouse events occurring on
either of two
```

```
different Frame objects.  It distinguishes between
the two
objects on the basis of the component name, and
displays
the coordinates of a mouse click on whichever
object
experienced the mouse click.

This program was tested using JDK 1.1.3 under
Win95.

Note that when the program starts, the two visual
components overlay one another.  It is necessary
to move
one of them in order to access the other.
**********************************************
********/

import java.awt.*;
import java.awt.event.*;

public class Event10 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class Event10
//================================================
=======//

//Subclass Frame in order to override the paint
method.
class MyFrame extends Frame{
  int xCoor;
  int yCoor;

  MyFrame(String name){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
```

```
    setSize(300,200);
    //Name used to distinguish between the two
objects
    setName(name);
  }//end constructor

  public void paint(Graphics g){
    //display coordinate information on the visual
object
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//===========================================
=======//

//The following class is used to instantiate a
graphical
// user interface object.
class GUI {
  public GUI(){//constructor
    //Create two visual objects of type MyFrame
and make
    // them visible. Name them Frame1 and Frame2.
    MyFrame myFrame1 = new MyFrame("Frame1");
    myFrame1.setVisible(true);

    MyFrame myFrame2 = new MyFrame("Frame2");
    myFrame2.setVisible(true);

    //Instantiate and register Listener object
which will
    // terminate the program when the user closes
either
    // window.
    WProc1 winProcCmd1 = new WProc1();
    myFrame1.addWindowListener(winProcCmd1);
```

```
    myFrame2.addWindowListener(winProcCmd1);

    //Instantiate and register Listener object
which will
    // process mouse events on either MyFrame
object.
    MouseProc mouseProcCmd =
                           new
MouseProc(myFrame1,myFrame2);
    myFrame1.addMouseListener(mouseProcCmd);
    myFrame2.addMouseListener(mouseProcCmd);
  }//end constructor
}//end class GUI definition
//=============================================
=======//

//This listener class monitors for mouse presses
and
// displays the coordinates of the mouse pointer
when the
// mouse is pressed.  The listener object
distinguishes
// between two different visual objects on the
basis of
// their component names and displays the
coordinate
// information on the visual object which
generated the
// mouse event.

class MouseProc extends MouseAdapter{
  //save references to the objects here
  MyFrame refToFrame1,refToFrame2;

  MouseProc(MyFrame inFrame1,MyFrame inFrame2)
{//construct
    refToFrame1 = inFrame1;//save references to
```

```
the frames
    refToFrame2 = inFrame2;
  }//end constructor

  //Override the mousePressed() method to respond
whenever
  // the mouse is pressed on one of the frame
objects.
  // Distinguish between the two frame objects
using the
  // component name and display the coordinates of
the
  // mouse on the correct object.
  public void mousePressed(MouseEvent e){
    if(
e.getComponent().getName().compareTo("Frame1") ==
0 ){
      //Get X and Y coordinates of mouse pointer
      // and store in the Frame object
      refToFrame1.xCoor = e.getX();
      refToFrame1.yCoor = e.getY();
      //display coordinate information
      refToFrame1.repaint();
    }else{
      //Get X and Y coordinates of mouse pointer
      //and store in the Frame object
      refToFrame2.xCoor = e.getX();
      refToFrame2.yCoor = e.getY();
      //display coordinate information
      refToFrame2.repaint();
    }//end if-else
  }//end mousePressed()
}//end class MouseProc
//=============================================
=======//

//The following listener is used to terminate the
```

```
program
// when the user closes either frame object. Note
that
// class extends the adapter class
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//=============================================
=======//
```

Second sample program

A single listener object can be registered to process events of a given class generated by two or more different visual objects.

Unique names can be assigned to the objects when they are instantiated.

When an event occurs, the code in the listener object can obtain the name of the visual object that generated the event and use it to distinguish among the objects.

The name of the object that generated the event can be determined using the statement shown in Listing 5 where

- **e** is the local name of the object passed into the low-level event-handling method, and
- **"Frame"** is a string being tested against the component name of a visual object that may have generated the event.

Changes relative to Event10

Although the program named **Event10** discussed above used two visual objects of the same type, there is no reason that the visual objects have to be of the same type, as long as all of the visual objects that share the **listener** object are capable of generating events of the event class for which the listener is designed.

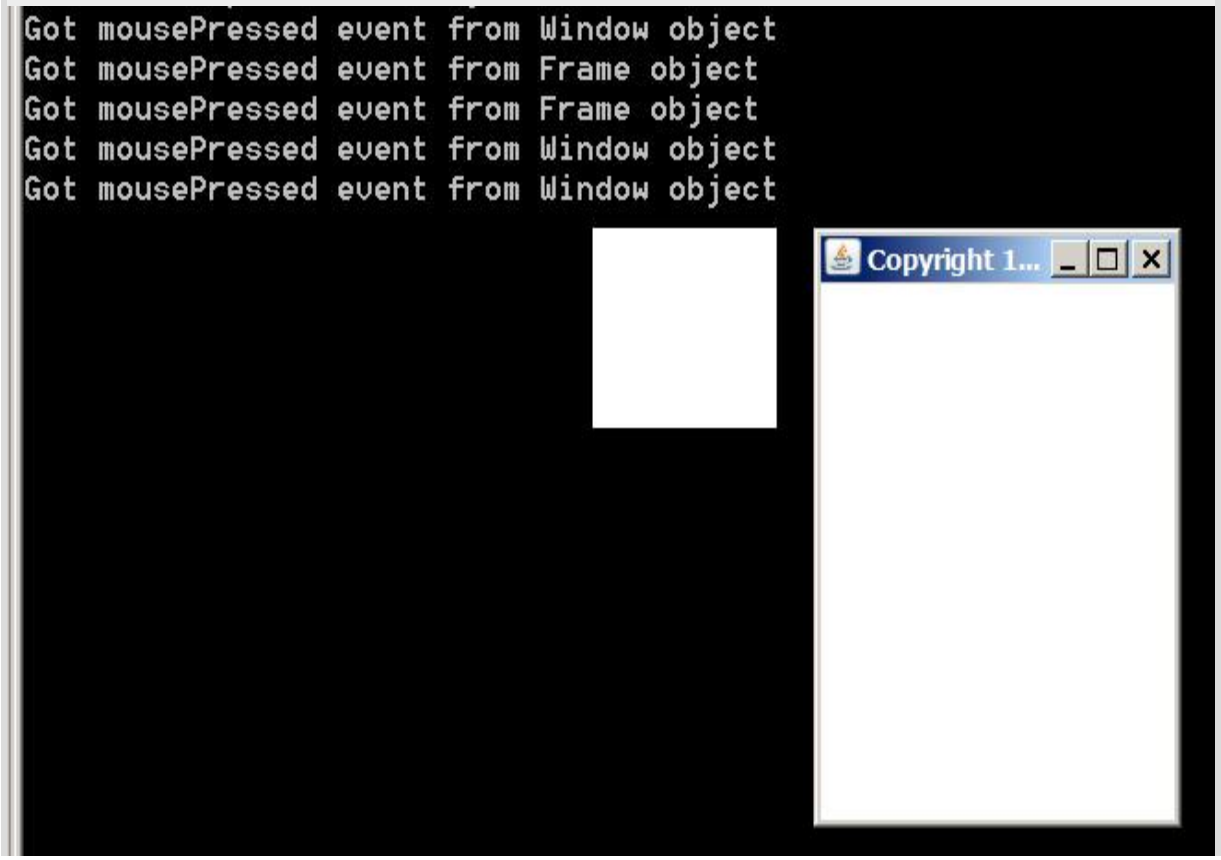
This is illustrated by the program named **Event11** *(see Listing 7)* that modifies the program named **Event10** to use

- a visual Frame object and
- a visual Window object,

instead of two **Frame** objects.

The screen output is shown in Figure 2 .

Note: Figure 2. Screen output from Event11.


Got mousePressed event from Window object
Got mousePressed event from Frame object
Got mousePressed event from Frame object
Got mousePressed event from Window object
Got mousePressed event from Window object
Copyright 1...

Note that the **Window** object, shown as the white square in the upper left of Figure 2 , doesn't have a banner that allows it to be physically moved with the mouse. Therefore, I called the **setBounds** method on both objects to separate them and to place them in a location on the screen where I could display the white **Window** object against the black background of the command-line window.

The ability to display the coordinates of mouse clicks was also removed for simplification of the program. Information about mouse clicks is displayed in the command-line window as shown in Figure 2 . The operation of the program is discussed in the comments in Listing 7 .

Note that an improved version of this program is also presented at the end of this lesson. The improved version does not require the establishment of source object names, and does not require the passing of references to the constructor for the Window listener in order to distinguish between the two sources. You should become familiar with the methodology used in the improved version as well as the methodology used in the version shown in Listing 7 .

Complete program listing for Event11

A complete listing of the program named **Event11** is shown in Listing 7 .

Note: Listing 7. The program named Event11.

```
/*File Event11.java Copyright 1997, R.G.Baldwin
This program is designed to be compiled and run
under
JDK 1.1
```

```
The program illustrates the sharing of a single
listener
object between two different visual components of
different types.

The program detects mouse events occurring on
either a
visual Frame object or on a visual Window object.
It
distinguishes between the two objects on the basis
of the
component name, and displays a message indicating
which
object generated the event.

Clicking inside the Frame object but outside the
Window
object produces the following message:

Got mousePressed event from Frame object

Clicking inside the Window object produces the
following
message:

Got mousePressed event from Window object

Closing the Frame object produces the following
message
and terminates the program:

Got windowClosing event from Frame object

These results were produced using JDK 1.1.3,
running under
Windows 95.
**********************************************
```

```
********/

import java.awt.*;
import java.awt.event.*;

public class Event11 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class Event11
//===========================================
=======//

//The following class is used to instantiate a
graphical
// user interface object.
class GUI {
  public GUI(){//constructor
    //Create a visual Frame object and name it
Frame
    Frame myFrame = new Frame();
    myFrame.setBounds(500,500,200,300);
    myFrame.setTitle("Copyright 1997,
R.G.Baldwin");
    myFrame.setName("Frame");
    myFrame.setVisible(true);

    //Create a visual Window object inside the
Frame
    // object and name it Window
    Window myWindow = new Window(myFrame);
    myWindow.setBounds(380,500,100,100);
    myWindow.setName("Window");
    myWindow.setVisible(true);

    //Instantiate and register a Listener object
which
```

```
    // will process mouse events on either the
Frame
    // object or the Window object.
    MouseProc mouseProcCmd = new MouseProc();
    myFrame.addMouseListener(mouseProcCmd);
    myWindow.addMouseListener(mouseProcCmd);

    //Instantiate and register a Listener object
which
    // will display a message and terminate the
program
    // when the user closes the Frame object
    WProc1 winProcCmd1 = new WProc1();
    myFrame.addWindowListener(winProcCmd1);
  }//end constructor
}//end class GUI definition
//===========================================
=======//

//This listener class monitors for mouse presses
and
// displays a message when a mousePressed() event
occurs on
// either the Frame object or the Window object.
The
// message identifies which visual object
generated
// the event. The listener object distinguishes
between
// the two visual objects on the basis of their
component
// names.

class MouseProc extends MouseAdapter{

  //Override the mousePressed() method to respond
whenever
```

```
  // the mouse is pressed on one of the visual
objects.
  public void mousePressed(MouseEvent e){
    if(
e.getComponent().getName().compareTo("Frame")

== 0 ){
      System.out.println(
               "Got mousePressed event from Frame
object");
    }//end if
    if(
e.getComponent().getName().compareTo("Window")

== 0 ){
      System.out.println(
              "Got mousePressed event from Window
object");
    }//end if
  }//end mousePressed()
}//end class MouseProc
//=============================================
=======//

//The following listener is used to display a
message and
// terminate the program when the user closes the
Frame
// object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.out.println(
              "Got windowClosing event from Frame
object");
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
```

```
//================================================
=======//
```

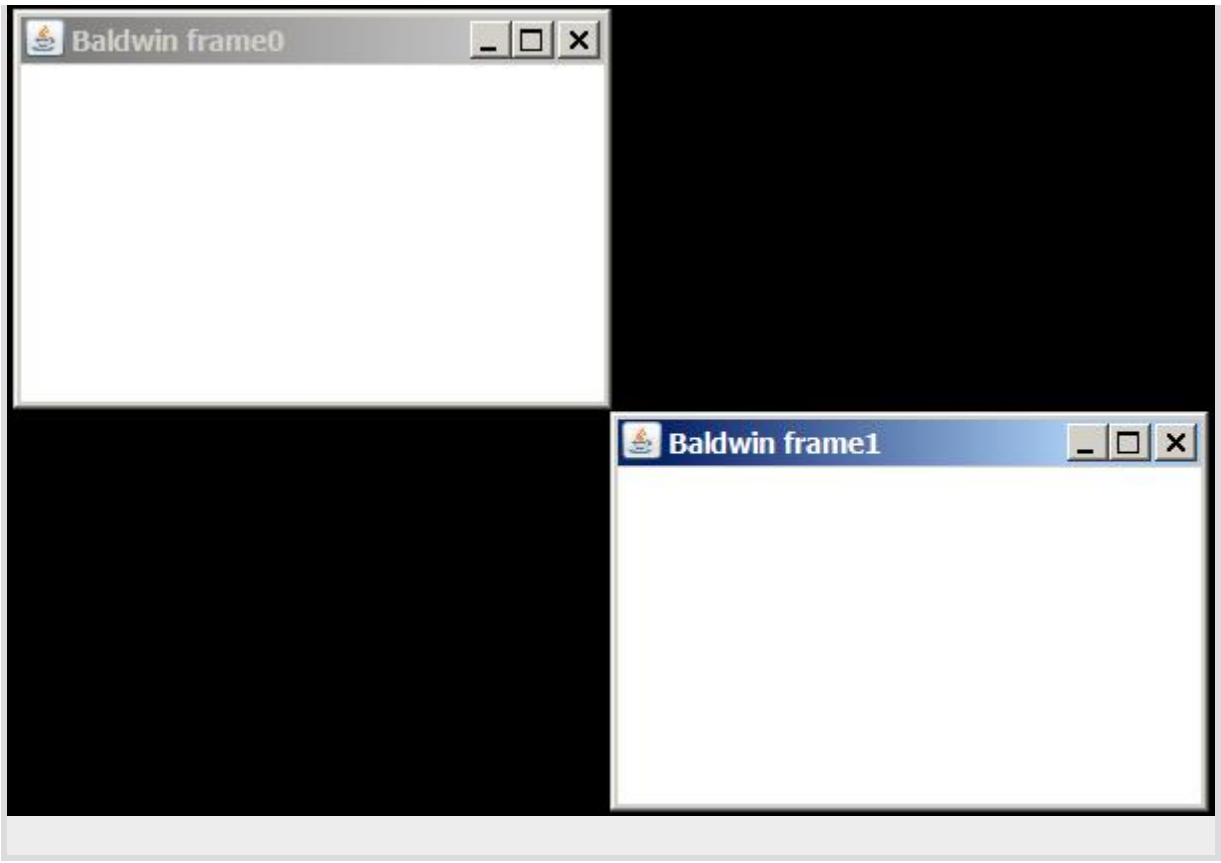
Review

Question 1

Write a Java application that displays two **Frame** objects on the screen. Each object has a width of 300 pixels and a height of 200 pixels.

One object is located in the upper left-hand corner of the screen. The top left-hand corner of the other object barely touches the bottom right-hand corner of the first object as shown in Figure 3 .

Note: Figure 3. Screen output for Question 1.



Make your name and the name of each object appear in the banner at the top of each object. Do not assign names to the objects.

Cause the two objects to share a single listener object to respond to mouse events.

Whenever the mouse is pressed internal to either object, the coordinates of the mouse pointer are displayed near the pointer on that object with the horizontal coordinate being displayed first followed by the vertical coordinate. The two coordinate values are separated by a comma and a space.

Also cause the two objects to share a single listener object that will terminate the program whenever the user clicks the "close" button on either object.

Make certain that your application terminates and returns control to the operating system when the user clicks on the "close" button in the upper right-hand corner of the object.

Solution - See the program in Listing 8 .

Note: Listing 8. Solution to Question 1.

```
/*File SampProg120.java from lesson 82
Copyright 1997, R.G.Baldwin

Without viewing the solution that follows, write a
Java
application that displays two Frame objects on the
screen.

Each object has a width of 300 pixels and a height
of 200
pixels.

One object is located in the upper left-hand
corner of the
screen.  The top left-hand corner of the other
object barely
touches the bottom right-hand corner of the first
object.

Make your name and the name of each object appear
in the
banner at the top of each object.

Do not assign names to the objects.

Cause the two objects to share a single Listener
object
```

```
to respond to mouse events.

Whenever the mouse is pressed internal to either
object,
the coordinates of the mouse pointer are displayed
near
the pointer on that object with the horizontal
coordinate
being displayed first followed by the vertical
coordinate.
The two coordinate values are separated by a comma
and a
space.

Also cause the two objects to share a single
Listener
object that will terminate the program whenever
the user
clicks the "close" button on either object.

Make certain that your application terminates and
returns
control to the operating system when the user
clicks on
the "close" button in the upper right-hand corner
of the
object.

*/
//=============================================
=========

import java.awt.*;
import java.awt.event.*;

public class SampProg120 {
  public static void main(String[] args){
```

```
    GUI gui = new GUI();
  }//end main
}//end class SampProg120
//-----------------------------------------------
---------

//Subclass Frame in order to override the paint
method.
class MyFrame extends Frame{
  int xCoor;
  int yCoor;

  MyFrame(){//constructor
    setTitle("Baldwin " + this.getName());
    setSize(300,200);
  }//end constructor

  public void paint(Graphics g){//override paint()
method
    //display coordinate information on the object
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//-----------------------------------------------
---------

class GUI {
  public GUI(){//constructor
    //Create two visual objects of type MyFrame,
specify
    // their locations, and make them visible.
    MyFrame myFrame1 = new MyFrame();
    myFrame1.setLocation(new Point(0,0));
    myFrame1.setVisible(true);

    MyFrame myFrame2 = new MyFrame();
```

```
    myFrame2.setLocation(new Point(300,200));
    myFrame2.setVisible(true);

    //Instantiate and register Listener object
which will
    // terminate the program when the user closes
either
    // window.
    WProc1 winProcCmd1 = new WProc1();
    myFrame1.addWindowListener(winProcCmd1);
    myFrame2.addWindowListener(winProcCmd1);

    //Instantiate and register Listener object
which will
    // process mouse events on either MyFrame
object.
    MouseProc mouseProcCmd =
      new MouseProc(myFrame1,myFrame2);
    myFrame1.addMouseListener(mouseProcCmd);
    myFrame2.addMouseListener(mouseProcCmd);
  }//end constructor
}//end class GUI definition
//-----------------------------------------------
---------

//This listener class monitors for mouse presses
and
// displays the coordinates of the mouse pointer
when the
// mouse is pressed.  The listener object
distinguishes
// between two different visual objects on the
basis of
// their component names and displays the
coordinate
// information on the visual object which
generated the
```

```
// mouse event.

class MouseProc extends MouseAdapter{
  //save references to the objects here
  MyFrame refToFrame1,refToFrame2;

  MouseProc(MyFrame inFrame1,MyFrame inFrame2)
{//constructor
    refToFrame1 = inFrame1;//save references to
the frames
    refToFrame2 = inFrame2;
  }//end constructor

  //Override the mousePressed() method to respond
whenever
  // the mouse is pressed on one of the frame
objects.
  // Distinguish between the two frame objects
using the
  // component name and display the coordinates of
the
  // mouse on the correct object.
  public void mousePressed(MouseEvent e){
     if( e.getComponent().getName().
          compareTo(refToFrame1.getName()) == 0)
    {
      refToFrame1.xCoor = e.getX();
      refToFrame1.yCoor = e.getY();
      //display coordinates on Frame1
      refToFrame1.repaint();
    }else{
      refToFrame2.xCoor = e.getX();
      refToFrame2.yCoor = e.getY();
      //display coordinates on Frame2
      refToFrame2.repaint();
    }//end if-else
  }//end mousePressed()
```

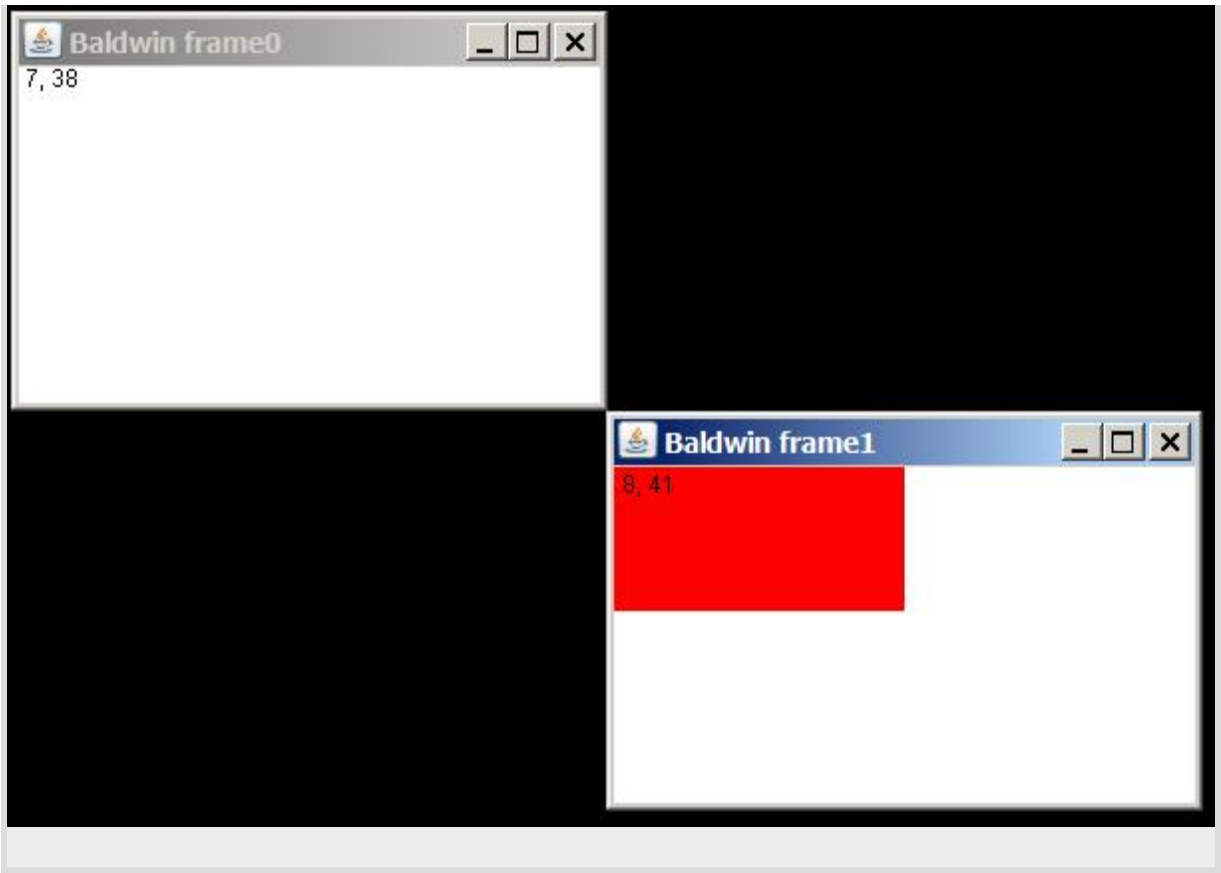
```
}//end class MouseProc
//-----------------------------------------------
---------

//The following listener is used to terminate the
program
// when the user closes either frame object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//-----------------------------------------------
---------
```

Question 2

Write a Java application that displays two **Frame** objects on the screen. Place a red **Panel** object in the upper-left quadrant of the second **Frame** object as shown in Figure 4 . *(Note that the red **Panel** is partially covered by the blue banner at the top of the **Frame** .)*

Note: Figure 4. Screen output for Question 2.



Each **Frame** object has a width of 300 pixels and a height of 200 pixels.

One **Frame** object is located in the upper left-hand corner of the screen. The top left-hand corner of the second **Frame** object barely touches the bottom right-hand corner of the first **Frame** object.

Make your name and the name of each **Frame** object appear in the banner at the top of each **Frame** object.

Cause the first **Frame** object and the **Panel** object to share a single listener object to respond to mouse events. Do not allow the second **Frame** object *(which contains the **Panel** object)* to share the listener object for mouse events.

Whenever the mouse is pressed internal to the first **Frame** object, or on the red portion of the **Panel** object, the coordinates of the mouse pointer are

displayed near the pointer on that object with the horizontal coordinate being displayed first followed by the vertical coordinate. The two coordinate values are separated by a comma and a space.

Whenever the mouse is pressed internal to the second **Frame** object, but not on the red **Panel** object, coordinate values are not displayed.

Also cause the two **Frame** objects to share a single listener object that will terminate the program whenever the user clicks the "close" button on either **Frame** object.

Make certain that your application terminates and returns control to the operating system when the user clicks on the "close" button in the upper right-hand corner of either **Frame** object.

Solution - See the program in Listing 9 .

Note: Listing 9. Solution to Question 2.

```
/*File SampProg121.java from lesson 82
Copyright 1997, R.G.Baldwin

Without viewing the solution that follows, write a
Java
application that displays two Frame objects on the
screen.
Place a red Panel object in the upper-left
quadrant of the
second Frame object.

Each Frame object has a width of 300 pixels and a
height
of 200 pixels.

One Frame object is located in the upper left-hand
```

```
corner
of the screen.  The top left-hand corner of the
second
Frame object barely touches the bottom right-hand
corner of
the first Frame object.

Make your name and the name of each Frame object
appear in
the banner at the top of each Frame object.

Cause the first Frame object and the Panel object
to
share a single Listener object to respond to mouse
events.
Do not allow the second Frame object to share the
Listener
object for mouse events.

Whenever the mouse is pressed internal to the
first Frame
object, or on the red portion of the Panel object,
the
coordinates of the mouse pointer are displayed
near
the pointer on that object with the horizontal
coordinate
being displayed first followed by the vertical
coordinate.
The two coordinate values are separated by a comma
and a
space.

Whenever the mouse is pressed internal to the
second Frame
object, but not on the red Panel object,
coordinate values
```

```
are not displayed.

Also cause the two Frame objects to share a single
Listener
object that will terminate the program whenever
the user
clicks the "close" button on either Frame object.

Make certain that your application terminates and
returns
control to the operating system when the user
clicks on
the "close" button in the upper right-hand corner
of either
Frame object.

*/
//=============================================
=========

import java.awt.*;
import java.awt.event.*;

public class SampProg121 {
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class SampProg121
//---------------------------------------------
---------

//Subclass Frame in order to override the paint
method.
class MyFrame extends Frame{
  int xCoor;
  int yCoor;
```

```
  MyFrame(){//constructor
    setTitle("Baldwin " + this.getName());
    setSize(300,200);
  }//end constructor

  public void paint(Graphics g){//override paint()
method
    //display coordinate information on the object
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//-----------------------------------------------
---------

//Subclass Panel in order to override the paint
method.
class MyPanel extends Panel{
  int xCoor;
  int yCoor;

  MyPanel(){//constructor
    setBounds(new Rectangle(0,0,150,100));
    setBackground(Color.red);
  }//end constructor

  public void paint(Graphics g){//override paint()
method
    //display coordinate information on the object
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//-----------------------------------------------
---------

class GUI {
```

```
  public GUI(){//constructor
    //Instantiate an object of type MyPanel
    MyPanel myPanel = new MyPanel();

    //Instantiate two objects of type MyFrame,
specify
    // their locations, and make them visible.
Place the
    // MyPanel object in the second MyFrame
object.
    MyFrame myFrame1 = new MyFrame();
    myFrame1.setLocation(new Point(0,0));
    myFrame1.setVisible(true);

    MyFrame myFrame2 = new MyFrame();
    myFrame2.setLayout(null);
    myFrame2.setLocation(new Point(300,200));
    myFrame2.add(myPanel);
    myFrame2.setVisible(true);

    //Instantiate and register a Listener object
which will
    // terminate the program when the user closes
either
    // window.
    WProc1 winProcCmd1 = new WProc1();
    myFrame1.addWindowListener(winProcCmd1);
    myFrame2.addWindowListener(winProcCmd1);

    //Instantiate and register a Listener object
which will
    // process mouse events on either the MyFrame
object
    // or the myPanel object.
    MouseProc mouseProcCmd =
      new MouseProc(myFrame1,myPanel);
    myFrame1.addMouseListener(mouseProcCmd);
```

```
    myPanel.addMouseListener(mouseProcCmd);

  }//end constructor
}//end class GUI definition
//-----------------------------------------------
---------

//This listener class monitors for mouse presses
and
// displays the coordinates of the mouse pointer
when the
// mouse is pressed.  The listener object
distinguishes
// between two different visual objects on the
basis of
// their component names and displays the
coordinate
// information on the visual object which
generated the
// mouse event.

class MouseProc extends MouseAdapter{
  //save references to the objects here
  MyFrame refToFrame1;
  MyPanel refToPanel;

  MouseProc(MyFrame inFrame1,MyPanel inPanel)
{//constructor
    refToFrame1 = inFrame1;//save references to
the frames
    refToPanel = inPanel;
  }//end constructor

  //Override the mousePressed() method to respond
whenever
  // the mouse is pressed on one of the frame
objects.
```

```
  // Distinguish between the two frame objects
using the
  // component name and display the coordinates of
the
  // mouse on the correct object.
  public void mousePressed(MouseEvent e){
     if( e.getComponent().getName().
          compareTo(refToFrame1.getName()) == 0)
    { //display coordinates on the Frame object
      refToFrame1.xCoor = e.getX();
      refToFrame1.yCoor = e.getY();
      refToFrame1.repaint();
    }else{//display coordinates on the Panel
object
      refToPanel.xCoor = e.getX();
      refToPanel.yCoor = e.getY();
      refToPanel.repaint();
    }//end if-else
  }//end mousePressed()
}//end class MouseProc
//-----------------------------------------------
---------

//The following listener is used to terminate the
program
// when the user closes either frame object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//-----------------------------------------------
---------
```

Question 3

Rewrite the program named **Event10** and eliminate the requirement to pass parameters to the constructors for the event listener objects.

Solution - See the program in Listing 10 .

Note: Listing 10. Solution to Question 3.

```
/*File Event10A.java Copyright 1997, R.G.Baldwin
Rewrite the program named Event10 and eliminate
the
requirement to pass parameters to the constructors
for the
event Listener objects.

This program is designed to be compiled and run
under
JDK 1.1

The program illustrates the sharing of a single
listener
object among two different visual components of
the same
type.

The program detects mouse events occurring on
either of two
different Frame objects.  It distinguishes between
the two
objects and displays the coordinates of a mouse
click on
whichever object experienced the mouse click.

These results were produced using JDK 1.1.3
```

```
running under
Windows 95.

Note that when the program starts, the two visual
components overlay one another.  It is necessary
to move
one of them in order to access the other.
*/
//=============================================
=======//

import java.awt.*;
import java.awt.event.*;

public class Event10A {
  public static void main(String[] args){
    //instantiate a Graphical User Interface
object
    GUI gui = new GUI();
  }//end main
}//end class Event10A
//=============================================
=======//

//Subclass Frame in order to override the paint
method.
class MyFrame extends Frame{
  int xCoor;
  int yCoor;

  MyFrame(){//constructor
    setTitle("Copyright 1997, R.G.Baldwin");
    setSize(300,200);
  }//end constructor

  public void paint(Graphics g){
    //display coordinate information on the visual
```

```
object
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//===========================================
=======//

//The following class is used to instantiate a
graphical
// user interface object.
class GUI {
  public GUI(){//constructor
    //Create two visual objects of type MyFrame
and make
    // them visible.
    MyFrame myFrame1 = new MyFrame();
    myFrame1.setVisible(true);

    MyFrame myFrame2 = new MyFrame();
    myFrame2.setVisible(true);

    //Instantiate and register Listener object
which will
    // terminate the program when the user closes
either
    // window.
    WProc1 winProcCmd1 = new WProc1();
    myFrame1.addWindowListener(winProcCmd1);
    myFrame2.addWindowListener(winProcCmd1);

    //Instantiate and register Listener object
which will
    // process mouse events on either MyFrame
object.
    MouseProc mouseProcCmd = new MouseProc();
    myFrame1.addMouseListener(mouseProcCmd);
```

```
    myFrame2.addMouseListener(mouseProcCmd);
  }//end constructor
}//end class GUI definition
//===========================================
=======//

//This listener class monitors for mouse presses
and
// displays the coordinates of the mouse pointer
when the
// mouse is pressed.  The listener object
distinguishes
// between two different visual objects and
displays the
// coordinate information on the visual object
which
// generated the mouse event.

class MouseProc extends MouseAdapter{
  //Override the mousePressed() method to respond
whenever
  // the mouse is pressed on one of the frame
objects.
  public void mousePressed(MouseEvent e){
    //Get X and Y coordinates of mouse pointer and
store in
    // the Frame object.  Distinguish between the
two
    // components on the basis of the source of
the event.
    // Note that the following two formulations
for X and Y
    // can be used to produce the same results in
this
    // situation.
    ((MyFrame)e.getComponent()).xCoor = e.getX();
    ((MyFrame)e.getSource()).yCoor = e.getY();
```

```
    //display coordinate information
    e.getComponent().repaint();
  }//end mousePressed()
}//end class MouseProc
//=============================================
=======//

//The following listener is used to terminate the
program
// when the user closes either frame object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//=============================================
=======//
```

Question 4

Rewrite the program named **Event11** and eliminate the requirement to pass parameters to the constructor for the listener objects.

Solution - See the program in Listing 11 .

Note: Listing 11. Solution to Question 4.

```
/*File Event11A.java Copyright 1997, R.G.Baldwin
Rewrite the program named Event11 and eliminate
the
```

```
requirement to pass parameters to the constructor
for the
listener objects.

This program is designed to be compiled and run
under
JDK 1.1

The program illustrates the sharing of a single
listener
object between two different visual components of
different
types.

The program detects mouse events occurring on
either a
visual Frame object or on a visual Window object.
It
distinguishes between the two objects and displays
a
message indicating which object generated the
event.

Clicking inside the Frame object but outside the
Window
object produces the following message:

Got mousePressed event from Frame object

Clicking inside the Window object produces the
following
message:

Got mousePressed event from Window object

Closing the Frame object produces the following
message and
```

```
terminates the program:

Got windowClosing event from Frame object

These results were produced using JDK 1.1.3
running under
Windows 95.
*/
//=============================================
=======//

import java.awt.*;
import java.awt.event.*;

public class Event11A {
  public static void main(String[] args){
    //instantiate a Graphical User Interface
object
    GUI gui = new GUI();
  }//end main
}//end class Event11A
//=============================================
=======//

//The following class is used to instantiate a
graphical
// user interface object.
class GUI {
  public GUI(){//constructor
    //Create a visual Frame object
    Frame myFrame = new Frame();
    myFrame.setSize(200,300);
    myFrame.setTitle("Copyright 1997,
R.G.Baldwin");
    myFrame.setVisible(true);

    //Create a visual Window object inside the
```

```
Frame object
    Window myWindow = new Window(myFrame);
    myWindow.setSize(100,100);
    myWindow.setVisible(true);

    //Instantiate and register a Listener object
which will
    // process mouse events on either the Frame
object or
    // the Window object.
    MouseProc mouseProcCmd = new MouseProc();
    myFrame.addMouseListener(mouseProcCmd);
    myWindow.addMouseListener(mouseProcCmd);

    //Instantiate and register a Listener object
which will
    // display a message and terminate the program
when the
    // user closes the Frame object
    WProc1 winProcCmd1 = new WProc1();
    myFrame.addWindowListener(winProcCmd1);
  }//end constructor
}//end class GUI definition
//=============================================
=======//

//This listener class monitors for mouse presses
and
// displays a message when a mousePressed() event
occurs on
// either the Frame object or the Window object.
The
// message identifies which visual object
generated the
// event. The listener object distinguishes
between the two
// visual objects.
```

```
class MouseProc extends MouseAdapter{

  //Override the mousePressed() method to respond
whenever
  // the mouse is pressed on one of the visual
objects.
  public void mousePressed(MouseEvent e){
    System.out.print("Got mousePressed event from
");
    if(e.getSource().toString().indexOf("Frame")
>= 0)
      System.out.println("Frame object");
    else
      System.out.println("Window object");
  }//end mousePressed()
}//end class MouseProc
//===================================================
=======//

//The following listener is used to display a
message and
// terminate the program when the user closes the
Frame
// object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.out.println(
              "Got windowClosing event from Frame
object");
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===================================================
=======//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java082 Sharing a Listener Object Among Visual Components
- File: Java082.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java084 Low-level and Semantic Events
An earlier lesson briefly discussed the differences between low-level events and semantic events as described in the JDK 1.1 documentation. This lesson takes another look at that topic through the use of programs which apply both low-level event handling and semantic event handling to the same set of visual components. This makes it possible to compare the two types of events in a more meaningful way.

Revised: Fri Jul 08 13:46:29 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- Overview
- Sample program for low-level events
- Sample program for low-level and semantic events
    - Focus events
    - Action events
    - ActionListener object
    - FocusListener object
    - MouseListener object
    - WindowListener object
    - More general comments about the program

- Review
    - Question 1
    - Question 2
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

Students in Prof. Baldwin's **ITSE 2317 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Graphic screen output for Event13.
- Figure 2. Sample text output from Event13.
- Figure 3. Graphic screen output for Event12.
- Figure 4. Screen output from SampProg200.
- Figure 5. Screen output from SampProg201.

Listings

- Listing 1. The program named Event13.
- Listing 2. The program named Event12.
- Listing 3. The solution to Question 1.
- Listing 4. The solution to Question 2.

Introduction

An earlier lesson briefly discussed the differences between *low-level* events and *semantic* events as described in the JDK 1.1 documentation.

This lesson takes another look at that topic through the use of programs that apply both *low-level* event handling and *semantic* event handling to the same set of visual components. This makes it possible to compare the two types of events in a more meaningful way.

Overview

Although the suite of *semantic* events is generally used for different purposes than the suite of *low-level* events, from a practical programming viewpoint, there is very little difference.

The primary difference appears to reside in the nature of the event object that is passed to the event handler when an event occurs.

Using the information in the event object, *low-level* events can gain access to the specific **Component** object that generated the event.

Given a *low-level* event object, the **getComponent** method of the **java.awt.event.ComponentEvent** class will return a reference to the actual object that generated the event.

Once that reference is available, there are literally dozens of methods of the **Component** class that can be invoked on the object, such as **getLocation, getLocationOnScreen, getMaximumSize, getMinimumSize, getName,** etc.

A sample program in a previous lesson invoked the **getName** method on such an object to determine which object among several objects generated a low-level mouse event.

A sample program that we will see later in this lesson invokes some of the other available methods on such a component object.

All *low-level* event classes are subclasses of the **java.awt.event.ComponentEvent** class, so the event handlers for all *low-level* events have access to the object that generated the event.

Semantic events, on the other hand, do not subclass the **ComponentEvent** class. Rather, they subclass the superclass of **ComponentEvent** making them siblings of **ComponentEvent** .

Because they do not subclass **ComponentEvent** , the event objects passed into semantic event handlers do not provide a way to obtain a reference to the object that generated the event, and therefore cannot invoke the methods of the **Component** class on that object.

Whether this is important or not depends on your needs. For example, if you needed to determine the location of the object that generated an event, you could determine that location by processing a *low-level* event but you *(probably)* could not determine that location by processing a *semantic* event. *(Never say never unless you want some ten-year-old programming genius to prove you wrong).*

Regardless of the ability to access the object that generated the event, the name of that object is readily available to the event handlers of both *low-level* and *semantic* events.

In both cases, the name of the object is encapsulated in the event object passed as a parameter and can be extracted or tested using methods of the **String** class. Often knowing the name of the object is sufficient to accomplish the desired result.

Sample program for low-level events

The program named **Event 13** shown in Listing 1 illustrates some of the capability of low-level events.

Note: Listing 1. The program named Event13.

```
/*File Event13.java Copyright 1997, R.G.Baldwin
Revised 03/09/98 to make it fit the page better.

This program is designed to be compiled and run
under JDK 1.1

The program demonstrates the object-manipulation
capability
of low-level events.

A Button object and a TextField object are placed
in a
Frame object.

A MouseListener object is instantiated and
registered to
monitor for low-level mousePressed() events on all
three
objects.  Whenever a mousePressed() event occurs,
the
Listener object obtains and displays several
pieces of
information about the object that generated the
event.
Although this demonstration program only obtains
and
displays information as a result of mousePressed()
events,
all of the methods of the Component class are
```

```
available for
use at that point.  Thus the code in the event
handler
method could also modify some of the attributes of
the
object that generated the event.

Finally, a WindowListener object is instantiated
and
registered to terminate the program when the user
closes
the Frame object.

Starting the program and then clicking
successively on the
Button, the TextField, and the interior of the
Frame
produces the following output.

Name = Button1
Parent's name = Frame
Location = java.awt.Point[x=4,y=23]
Minimum Size =
java.awt.Dimension[width=54,height=21]
Size = java.awt.Dimension[width=192,height=21]

Name = TextField1
Parent's name = Frame
Location = java.awt.Point[x=4,y=275]
Minimum Size =
java.awt.Dimension[width=104,height=21]
Size = java.awt.Dimension[width=192,height=21]

Name = Frame
No parent name available at this level
Location = java.awt.Point[x=0,y=0]
Minimum Size =
```

```
java.awt.Dimension[width=112,height=69]
Size = java.awt.Dimension[width=200,height=300]


These results were produced using JDK 1.1.3, under
Win95.
**********************************************
********/

import java.awt.*;
import java.awt.event.*;

public class Event13 {
  public static void main(String[] args){
    //instantiate a Graphical User Interface
object
    GUI gui = new GUI();
  }//end main
}//end class Event13
//=============================================
=======//

class GUI {
  public GUI(){//constructor
    //Create a visual TextField object
    TextField myTextField = new TextField("Initial
String");
    myTextField.setName("TextField1");

    //Create a visual Button object
    Button myButton = new Button("Click me");
    myButton.setName("Button1");

    //Create a visual Frame object
    Frame myFrame = new Frame();
    myFrame.setSize(200,300);
    myFrame.setTitle("Copyright 1997,
```

```
R.G.Baldwin");
    myFrame.setName("Frame");

    //Add the Button and the TextField to the
Frame object
    myFrame.add("North",myButton);
    myFrame.add("South",myTextField);
    myFrame.setVisible(true);

    //Instantiate and register a MouseListener
object which
    // will process mouse events on the Frame
object, the
    // Button object, and the TextField object.
    MouseProc mouseProcCmd = new MouseProc();
    myFrame.addMouseListener(mouseProcCmd);
    myTextField.addMouseListener(mouseProcCmd);
    myButton.addMouseListener(mouseProcCmd);

    //Instantiate and register a Listener object
which will
    // terminate the program when the user closes
the
    // Frame object
    WProc1 winProcCmd1 = new WProc1();
    myFrame.addWindowListener(winProcCmd1);
  }//end constructor
}//end class GUI definition
//=============================================
=======//

//Low-level event monitor.
// This listener class monitors for low-level
// mousePressed() events. Whenever  mousePressed()
event
// occurs,  event handler obtains and displays
several
```

```
// pieces of information about  object that
generated
//  event.

class MouseProc extends MouseAdapter{
  public void mousePressed(MouseEvent e){
    System.out.println(
                    "Name = " +
e.getComponent().getName());
    try{
      System.out.println("Parent's name = " +

e.getComponent().getParent().getName());
    }catch(NullPointerException exception){
      System.out.println(
                  "No parent name available at this
level");
    }//end try/catch
    System.out.println("Location = " +

e.getComponent().getLocation().toString());
    System.out.println("Minimum Size = " +

e.getComponent().getMinimumSize().toString());
    System.out.println("Size = " +

e.getComponent().getSize().toString());
    System.out.println();//blank line
  }//end mousePressed()
}//end class MouseProc
//=============================================
=======//

//The following listener class is used to
terminate the
// program when the user closes the Frame object.
class WProc1 extends WindowAdapter{
```

```
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===========================================
=======//
```

The graphic screen output produced by this program is shown in Figure 1 .

Note: Figure 1. Graphic screen output for Event13.



A **Button** object and a **TextField** object are placed at the top and bottom of a **Frame** object.

A **MouseListener** object is instantiated and registered to monitor for low-level **mousePressed** events on all three objects.

Whenever a **mousePressed** event occurs, the listener object obtains and displays several pieces of information about the object that generated the event. Some samples of the text information produced by this program are shown in Figure 2 .

Note: Figure 2. Sample text output from Event13.

```
Name = Button1
Parent's name = Frame
Location = java.awt.Point[x=4,y=28]
Minimum Size =
java.awt.Dimension[width=62,height=23]
Size = java.awt.Dimension[width=192,height=23]

Name = Frame
No parent name available at this level
Location = java.awt.Point[x=0,y=0]
Minimum Size =
java.awt.Dimension[width=130,height=78]
Size = java.awt.Dimension[width=200,height=300]

Name = TextField1
Parent's name = Frame
Location = java.awt.Point[x=4,y=273]
Minimum Size =
java.awt.Dimension[width=122,height=23]
Size = java.awt.Dimension[width=192,height=23]
```

Although this demonstration program only obtains and displays information as a result of **mousePressed** events, all of the methods of the **Component** class are available for use at that point. Thus the code in the event handler

method could also modify some of the attributes of the object that generated the event.

Finally, a **WindowListener** object is instantiated and registered to terminate the program when the user closes the **Frame** object.

Additional information about the program is contained in the comments in Listing 1 .

Sample program for low-level and semantic events

The sample program named **Event 12** provides both low-level and semantic event handling for the same three components as the previous program. Listing 2 shows the program, which is fairly long, but it is also fairly repetitive.

Note: Listing 2. The program named Event12.

```
/*File Event12.java Copyright 1997, R.G.Baldwin
Revised 03/09/98 to fit the page better.
Revised 02/13/04 to correct an error in the
comments.

This program is designed to be compiled and run
under JDK 1.1

The program supports experimentation with low-
level events
and semantic events.

A Button object and a TextField object are placed
in a
Frame object.
```

```
An ActionListener object is instantiated and
registered to
monitor for semantic actionPerformed()events on
the Button
and the TextField.

An actionPerformed() event can be generated on a
TextField
by pressing the Enter key while the TextField
object has
the focus.

An actionPerformed() event can be generated by a
Button by
clicking on it with the mouse.

An action event cannot be generated by a Frame
object.

Whenever an actionPerformed() event occurs, the
Listener
object invokes the getActionCommand() method on
the object
to obtain the "command name".

The getActionCommand() method returns the "command
name"
associated with the action as a String.  The
string is
displayed.  As it turns out, the "command name"
associated
with a Button is simply the text, caption, or
label on the
button.  The "command name" associated with a
TextField is
the current text content of the TextField object.
```

```
The ActionEvent object passed to the
actionPerformed()
method includes the name of the component which
can be
used in a conditional test based on the indexOf()
method
of the String class to identify the component that
generated the event.

Each time the actionPerformed() method is invoked,
code in
the body of the method uses the indexOf() method
to
identify the component that generated the event
and
displays a message identifying that component.

A FocusListener object is instantiated and
registered to
monitor for low-level focusGained() and
focusLost() events
on the Button and the TextField.

Whenever a focusGained() event occurs, a message
is
displayed identifying the object which gained the
focus.
Likewise, whenever a focusLost() event occurs, a
message is
displayed identifying the object which lost the
focus.  The
object that gained or lost focus is identified by
performing conditional tests on the FocusEvent
object
passed in as a parameter.

A MouseListener object is instantiated and
```

registered to
monitor for low-level mousePressed() events on all three
objects. The Listener object differentiates among the
three on the basis of the component name assigned to each
object. The approach used to obtain the component name in
this program uses the indexOf() method of the String class
on the MouseEvent object. This is a somewhat less complex
approach than the approach used to obtain the component
name for a mousePressed() event in an earlier lesson. When
a mousePressed() event occurs on any of the three visual
objects, the Listener object displays a message identifying
the object that generated the event.

Finally, a WindowListener object is instantiated and
registered to terminate the program when the user closes
the Frame object.

Typical outputs from the program follow:

Clicking the mouse inside the frame but outside of both the
TextField and the Button produces the following output:

```
Got mousePressed event from Frame object


Clicking the mouse on the TextField when the
Button has the
focus produces the following output:

Got mousePressed event from TextField1 object
Got focusLost event from Button1 object
Got focusGained event from TextField1 object


Pressing the Enter key when the TextField has the
focus
produces the following output:

e.getActionCommand() = Initial String
Got actionPerformed event from TextField1 object


Clicking the mouse on the Button when the
TextField has the
focus produces the following output:

Got mousePressed event from Button1 object
Got focusLost event from TextField1 object
Got focusGained event from Button1 object
e.getActionCommand() = Click me
Got actionPerformed event from Button1 object


These results were produced using JDK 1.1.3 under
Win95.
**********************************************
********/

import java.awt.*;
```

```
import java.awt.event.*;

public class Event12 {
  public static void main(String[] args){
    //instantiate a Graphical User Interface
object
    GUI gui = new GUI();
  }//end main
}//end class Event12
//=============================================
=======//

//The following class is used to instantiate a
graphical
// user interface object.
class GUI {
  public GUI(){//constructor
    //Create a visual TextField object
    TextField myTextField =
                            new TextField("Initial
String");
    myTextField.setName("TextField1");

    //Create a visual Button object
    Button myButton = new Button("Click me");
    myButton.setName("Button1");

    //Create a visual Frame object and name it
Frame
    Frame myFrame = new Frame();
    myFrame.setSize(200,300);
    myFrame.setTitle("Copyright 1997,
R.G.Baldwin");
    myFrame.setName("Frame");

    //Add the Button and the TextField to the
Frame object
```

```
    myFrame.add("North",myButton);
    myFrame.add("South",myTextField);
    myFrame.setVisible(true);

    //Instantiate and register an ActionListener
object
    // which will monitor for action events on the
    // TextField and the Button.
    ActionProc actionProcCmd = new ActionProc();
    myTextField.addActionListener(actionProcCmd);
    myButton.addActionListener(actionProcCmd);

    //Instantiate and register a FocusListener
object which
    // will monitor for focus events on the
TextField and
    // the Button.
    FocusProc focusProcCmd = new FocusProc();
    myTextField.addFocusListener(focusProcCmd);
    myButton.addFocusListener(focusProcCmd);

    //Instantiate and register a MouseListener
object which
    // will process mouse events on the Frame
object, the
    // Button object, or the TextField object.
    MouseProc mouseProcCmd = new MouseProc();
    myFrame.addMouseListener(mouseProcCmd);
    myTextField.addMouseListener(mouseProcCmd);
    myButton.addMouseListener(mouseProcCmd);

    //Instantiate and register a Listener object
which will
    // terminate the program when the user closes
the Frame
    // object
    WProc1 winProcCmd1 = new WProc1();
```

```
    myFrame.addWindowListener(winProcCmd1);
  }//end constructor
}//end class GUI definition
//===========================================
=======//

//Semantic event monitor.
// This ActionListener class is used to
instantiate a
// Listener object that monitors for action events
on the
// TextField and the Button.  Whenever an
actionPerformed()
// event occurs, it displays the ActionCommand and
the
// identification of the component that generated
the
// event. The listener object distinguishes
between the
// components on the basis of their component
names which
// are embedded in the object passed in as a
parameter
// when an event occurs.

class ActionProc implements ActionListener{
  public void actionPerformed(ActionEvent e){
    System.out.println("e.getActionCommand() = " +

e.getActionCommand());

    if( e.toString().indexOf("on TextField1") !=
-1 ){
      System.out.println(
       "Got actionPerformed event from TextField1
object");
    }//end if
```

```
    if( e.toString().indexOf("on Button1") != -1 )
{
      System.out.println(
          "Got actionPerformed event from Button1
object");
    }//end if
  }//end actionPerformed()
}//end class ActionProc
//===================================================
=======//

//Low-level event monitor.
// This FocusListener class is used to instantiate
a
// Listener object that monitors for focus events
on the
// TextField and the Button.  Whenever a
focusLost() or
// focusGained() event occurs, it displays the
// identification of the component that generated
the
// event.  The listener object distinguishes
between the
// components on the basis of their component
names which
// are embedded in the object passed in as a
parameter when
// an event occurs.

class FocusProc implements FocusListener{
  public void focusGained(FocusEvent e){
    if( e.toString().indexOf("on TextField1") !=
-1 ){
      System.out.println(
           "Got focusGained event from TextField1
object");
```

```
    }//end if

    if( e.toString().indexOf("on Button1") != -1 )
{
      System.out.println(
              "Got focusGained event from Button1
object");
    }//end if
  }//end focusGained()

  public void focusLost(FocusEvent e){
    if( e.toString().indexOf("on TextField1") !=
-1 ){
      System.out.println(
             "Got focusLost event from TextField1
object");
    }//end if
    if( e.toString().indexOf("on Button1") != -1 )
{
      System.out.println(
                "Got focusLost event from Button1
object");
    }//end if
  }//end focusLost()
}//end class FocusProc
//=============================================
=======//

//Low-level event monitor.
// This listener class monitors for mouse presses
and
// displays a message when a mousPressed() event
occurs on
// the Frame object, the Button object, or the
TextField
// object.  The message identifies the component
that
```

```
// generated the event. The listener object
distinguishes
// between the components on the basis of their
component
// names which are embedded in the object passed
in as a
// parameter when an event occurs.

class MouseProc extends MouseAdapter{
  public void mousePressed(MouseEvent e){
    if( e.toString().indexOf("on Frame") != -1 ){
      System.out.println(
                "Got mousePressed event from Frame
object");
    }//end if

    if( e.toString().indexOf("on TextField1") !=
-1 ){
      System.out.println(
          "Got mousePressed event from TextField1
object");
    }//end if

    if( e.toString().indexOf("on Button1") != -1 )
{
      System.out.println(
              "Got mousePressed event from Button1
object");
    }//end if
  }//end mousePressed()
}//end class MouseProc
//=================================================
=======//

//The following listener is used to display a
message and
// terminate the program when the user closes the
```

```
Frame
// object.
class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===========================================
=======//
```

The graphic screen output for this program is shown in Figure 3 .

Note: Figure 3. Graphic screen output for Event12.



As before, a **Button** object and a **TextField** object are placed at the top and bottom respectively of a **Frame** object.

Semantic event handling is provided for **Action** events. *Low-level* event handling is provided on the same components for **mousePressed** events and **Focus** events.

Focus events

In Java, a component has the *focus* when it can accept keyboard input. There are a variety of ways by which the focus can move from one component to another. Whenever it moves, one component generates a **focusLost** event and the other component generates a **focusGained** event.

There are also some issues involving *temporary* and *permanent* changes in focus which aren't discussed here, but can be found in the JDK 1.1 documentation.

On the basis of the above description, you should see that there are many different types of components that can generate a focus event. Any component that can gain the focus can generate such an event.

There are some components such as **Button** objects and **TextField** objects that automatically gain the focus when they are clicked on by the mouse. There are other components such as **Label** objects that do not automatically gain the focus when they are clicked on by the mouse. However, even these components can gain the focus by *requesting* it. We will investigate this in more detail in a future lesson.

Action events

An action event can also be generated by many different types of components. For example, clicking a button or pressing the *Enter* key while a **TextField** object has the focus will generate an action event. The terminology derives from the notion that those user actions that generate action events are messages to the program to take the specific action indicated by the nature of the component.

For example, if a button is labeled **Exit** and it is clicked by the user, that means that the user is expecting an action that can be interpreted as *exit* in that context.

ActionListener object

In this program an **ActionListener** object is instantiated and registered to monitor for semantic **actionPerformed** events on the **Button** and the **TextField** .

When an **actionPerformed** event is generated, certain information regarding the event is encapsulated into an object that is passed to the **actionPerformed** method of the listener object. This information includes what the JDK 1.1 documentation refers to as a *command name* . This information can be accessed by the code in the method by invoking the **getActionCommand** method on the object. In this program, the *command name* is accessed and displayed on the screen.

As it turns out, the "command name" associated with a **Button** is simply the *text* , *caption* , or *label* on the button *(whatever you choose to call it).* The "command name" associated with a **TextField** is the current text content of the **TextField** object.

This information would have different uses for different components in different situations. For example, it might be used to distinguish among several buttons if the captions on the buttons were not allowed to change during the execution of the program. It might be used to extract user input from a **TextField** object.

The object of type **ActionEvent** passed to the **actionPerformed** method also includes the name of the component, which can be used in a conditional test to identify the component that generated the event. One way to do this is through use of the **indexOf** method of the **String** class to determine if a given component name is included in a specific object.

In this program, each time the **actionPerformed** method is invoked, code in the body of the method uses the **indexOf** method to identify the component that generated the event and displays a message identifying that component.

FocusListener object

A **FocusListener** object is instantiated and registered to monitor for *low-level* **focusGained** and **focusLost** events on the **Button** and the **TextField** .

Whenever a **focusGained** event occurs, a message is displayed identifying the object that gained the focus.

Likewise, whenever a **focusLost** event occurs, a message is displayed identifying the object that lost the focus.

The object that gained or lost the focus is identified by performing conditional tests on the **FocusEvent** object passed in as a parameter in the same manner that the **ActionEvent** object is used for action events.

MouseListener object

A **MouseListener** object is instantiated and registered to monitor for *low-level* **mousePressed** events on all three objects. *(Note that there are numerous other low-level mouse events that could be monitored but they were omitted in for the sake of simplicity.)*

The **MouseListener** object differentiates among the three objects *(**Frame** , **Button** , and **TextField***) on the basis of the component name assigned to each object when it is instantiated.

At this point, it would probably be worthwhile to point out that JDK 1.1 Beta 3 does not require the programmer to assign unique names to components when they are instantiated. Assigned component names can be duplicated among components. *(I wonder if this is a bug? Other products*

such as Visual Basic and Delphi prohibit the programmer from assigning duplicate names to components.)

If the programmer does not assign names to the components when they are instantiated, they are automatically assigned by the system, and are probably unique. The names that are automatically assigned have the format *frame0*, *frame1*, *frame2* , etc., with the main body of the name identifying the type of component, and the digit at the end being assigned in the order in which the components are instantiated. A clever programmer should be able to find a way to obtain and make use of those names as an alternative to assigning her own names.

The approach used to obtain the component name in this program uses the **indexOf** method of the **String** class on the **MouseEvent** object. *(This is a slightly less complex approach than the approach used to obtain the component name for a **mousePressed** event in an earlier lesson which went all the way back to the component object and invoked the **getName** method.)*

When a **mousePressed** event occurs on any of the three visual objects, the **MouseListener** object displays a message identifying the object that generated the event.

WindowListener object

Finally, a **WindowListener** object is instantiated and registered to terminate the program when the user closes the **Frame** object.

More general comments about the program

The screen output for a number of different user actions is shown in the comments at the beginning of the program. This material illustrates how the *action*, *focus*, and *mousePressed* events behave during normal user interaction with the program.

In order to maintain simplicity, the response to events in this program is limited to simply displaying information. Obviously, once control is within an event handler, more significant behavior in response to an event can be programmed.

This program illustrates the fact that a single user action can cause many different types of events to be generated.

One thing to notice in particular is that since this program was not designed to manipulate the objects that generated the *low-level* events, there is very little difference in the handling of *low-level* events and *semantic* events. However, the handling would have been significantly different if the capability to manipulate the objects that generated the events had been exercised *(as in the previous program).*

Review

Question 1

Using only low-level events, write a Java application that originally displays a **Frame** object containing a **Button** object at the top and a **TextField** object at the bottom. Cause the **TextField** to have red letters on a yellow background as shown in Figure 4 .

When you click on the **TextField** object, it disappears. When you click on the **Button** object, the TextField object reappears.

When you click on the close button in the upper right-hand corner of the **Frame** object, the program terminates and control is properly returned to the operating system.

Note: Figure 4. Screen output from SampProg200.



See the solution in Listing 3 .

Note: Listing 3. The solution to Question 1.

```
/*From lesson 84

Note that this appears to be a duplicate of
SampProg122.

 Copyright 1997, R.G.Baldwin

 Without viewing the following solution, write a
Java
 application that originally displays a Frame
object
 containing a button at the top and a TextField
object
 at the bottom.

 Cause the TextField to have red letters on a
yellow
 background.

 When you click on the TextField object, it
disappears.
 When you click on the Button object, the
```

```
TextField object
 reappears.

 Use only low level events.

 When you click on the close button in the upper
right-hand
 corner of the Frame object, the program
terminates and
 control is properly returned to the operating
system.

//=================================================
=========
 */

 import java.awt.*;
 import java.awt.event.*;

 public class SampProg200 {
   public static void main(String[] args){
     GUI gui = new GUI();
   }//end main
 }//end class SampProg200

//=================================================
=========

 class GUI {
   public GUI(){//constructor
     //Create a visual TextField object
     TextField myTextField = new
TextField("Initial String");
     myTextField.setName("TextField1");
     myTextField.setBackground(Color.yellow);
     myTextField.setForeground(Color.red);
```

```
     //Create a visual Button object
     Button myButton = new Button("Click me");
     myButton.setName("Button1");

     //Create a visual Frame object
     Frame myFrame = new Frame();
     myFrame.setSize(300,100);
     myFrame.setTitle("Copyright 1997,
R.G.Baldwin");

     //Add the Button and the TextField to the
Frame object
     myFrame.add("North",myButton);
     myFrame.add("South",myTextField);
     myFrame.setVisible(true);

     //Instantiate and register a MouseListener
object which
     // will process mouse events on the Button
object, and
     // the TextField object.
     MouseProc mouseProcCmd = new MouseProc(
       myButton,myTextField);
     myTextField.addMouseListener(mouseProcCmd);
     myButton.addMouseListener(mouseProcCmd);

     //Instantiate and register a Listener object
which will
     // terminate the program when the user closes
the
     // Frame object
     myFrame.addWindowListener(new WProc1());
   }//end constructor
 }//end class GUI definition

//=============================================
=========
```

```
 //Low-level event monitor.
 // This listener class monitors for low-level
mousePressed()
 // events. Whenever a mousePressed() event
occurs, the
 // event handler determines which object was the
source of
 // the event and takes the appropriate action.

 class MouseProc extends MouseAdapter{
   Button refToButton = null;
   TextField refToTextField = null;
   String refToButtonName = null;
   String refToTextFieldName = null;

   public MouseProc(//constructor
         Button inRefToButton, TextField
inRefToTextField){
     refToButton = inRefToButton;
     refToTextField = inRefToTextField;
     refToButtonName = inRefToButton.getName();
     refToTextFieldName =
inRefToTextField.getName();
   }//end constructor

   public void mousePressed(MouseEvent e){

if(e.getComponent().getName().compareTo(refToTextF
ieldName) == 0)
       refToTextField.setVisible(false);

if(e.getComponent().getName().compareTo(refToButto
nName) == 0)
       refToTextField.setVisible(true);
   }//end mousePressed()
 }//end class MouseProc
```

```
//===========================================
====================

 //The following listener class is used to
terminate the
 // program when the user closes the Frame object.
 class WProc1 extends WindowAdapter{
   public void windowClosing(WindowEvent e){
     System.exit(0);
   }//end windowClosing()
 }//end class WProc1

//===========================================
====================
```

Question 2

Using a mixture of low-level and semantic events, write a Java application that originally displays a **Frame** object containing a **Button** object at the top and a **TextField** object at the bottom. Cause the **TextField** to have red letters on a yellow background as shown in Figure 5 .

When you click on the **TextField** object, it disappears. When you click on the **Button** object, the **TextField** object reappears.

When you click on the close button in the upper right-hand corner of the **Frame** object, the program terminates and control is properly returned to the operating system.

Note: Figure 5. Screen output from SampProg201.



See the solution in Listing 4 .

Note: Listing 4. The solution to Question 2.

```
/*From lesson 84

Note that this appears to be a duplicate of
SampProg123
 Copyright 1997, R.G.Baldwin
 */

 import java.awt.*;
 import java.awt.event.*;

 public class SampProg201 {
   public static void main(String[] args){
     GUI gui = new GUI();
   }//end main
 }//end class SampProg201

//=============================================
=========

 class GUI {
   public GUI(){//constructor
     //Create a visual TextField object
```

```
    TextField myTxtField = new TextField("Initial
String");
    myTxtField.setBackground(Color.yellow);
    myTxtField.setForeground(Color.red);

    //Create a visual Button object
    Button myButton = new Button("Click me");

    //Create a visual Frame object
    Frame myFrame = new Frame();
    myFrame.setSize(300,100);
    myFrame.setTitle("Copyright 1997,
R.G.Baldwin");

    //Add the Button and the TextField to the
Frame object
    myFrame.add("North",myButton);
    myFrame.add("South",myTxtField);
    myFrame.setVisible(true);

    //Instantiate and register a MouseListener
object which
    // will process mouse events on the TextField
object.
    myTxtField.addMouseListener(new
MouseProc(myTxtField));

    //Instantiate and register an ActionListener
object
    // which will process action events on the
Button
    // object.
    myButton.addActionListener(
      new MyActionProcessor(myTxtField));

    //Instantiate and register a Listener object
which will
```

```
     // terminate the program when the user closes
the
     // Frame object
     myFrame.addWindowListener(new WProc1());
   }//end constructor
 }//end class GUI definition

//===================================================
=========

 //Low-level event monitor.
 // This listener class monitors for low-level
 // mousePressed() events.
 class MouseProc extends MouseAdapter{
   TextField refToTextField = null;

   public MouseProc(TextField inRefToTextField){
     refToTextField = inRefToTextField;
   }//end constructor

   public void mousePressed(MouseEvent e){
     refToTextField.setVisible(false);
   }//end mousePressed()
 }//end class MouseProc

//===================================================
=========
 //Semantic event monitor.
 // This listener class monitors for semantic
action events.

 class MyActionProcessor implements
ActionListener{
   TextField refToTextField = null;

   MyActionProcessor(TextField inRefToTextField)
{//construct
```

```
    refToTextField = inRefToTextField;
  }//end constructor

  public void actionPerformed(ActionEvent e){
    refToTextField.setVisible(true);
  }//end overridden actionPerformed method

}//end class MyActionProcessor



//===================================================
=========

 //The following listener class is used to
terminate the
 // program when the user closes the Frame object.
 class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
 }//end class WProc1

//===================================================
=========
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java084 Low-level and Semantic Events

- File: Java084.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java085 Swing, New Event Types in Swing
Although event handling using Swing components is the same as event handling using AWT components, the Swing classes provide a number of new event types. This lesson briefly discusses all of the new event types, and provides two different sample programs that illustrate event handling with one of the new event types.

Revised: Sat Jul 09 16:44:27 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- New event types in Swing
- First sample program
    - Interesting code fragments for SwingEvent10
    - Program listing for SwingEvent10
- Second sample program
    - Interesting code fragments for SwingEvent11
    - Program listing for SwingEvent11
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

Students in Prof. Baldwin's **Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Listener interfaces defined in Swing.
- Figure 2. Event classes defined in Swing.
- Figure 3. Graphic screen output for SwingEvent10.
- Figure 4. Text screen output from moving the JFrame.
- Figure 5. Information regarding the content pane.
- Figure 6. Methods of the AncestorListener interface.
- Figure 7. Graphic screen output for SwingEvent11.
- Figure 8. Text output from clicking buttons.

Listings

- Listing 1. Import directives for SwingEvent10.
- Listing 2. The main method for SwingEvent10.
- Listing 3. Beginning of the class named GUI.
- Listing 4. Definition of the AncestorAdded method.

- Listing 5. Remaining two methods of the AncestorListener interface.
- Listing 6. Complete listing of program named SwingEvent10.
- Listing 7. The constructor for the GUI class.
- Listing 8. Definition of the ancestor listener class.
- Listing 9. Definition of the action listener class.
- Listing 10. Complete listing of program named SwingEvent11.

Introduction

Although event handling using **Swing** components is essentially the same as event handling using **AWT** components, the **Swing** classes provide a number of new event types.

This lesson will briefly discuss the new event types *(as of Swing 1.0.1)* , and will provide two sample programs that illustrate event handling with one of the new event types.

One of the sample programs will also illustrate an important new aspect of **Swing** wherein every component is also a container. In this case, we will build a pyramid of **Swing JButton** objects where each **JButton** object is contained in the one below it with the bottom **JButton** object being contained in a **JFrame** object. *(See Figure 7 .)*

We will then illustrate how these **JButton** objects respond to action events and ancestor events. Action events come to us from the **AWT** whereas ancestor events are new to **Swing** .

New event types in Swing

One of the easiest ways to identify the new event types in **Swing** is to take a look at the listener interfaces defined in **Swing** . Another way is to take a look at the event classes defined in **Swing** .

Figure 1 shows a list of the listener interfaces defined in the **com.sun.java.swing.event** package of **Swing 1.0.1** . *(Note that sometime between 1997 and 2016, all Swing components were moved into the standard Java library in the javax.swing package.)*

Note: Figure 1. Listener interfaces defined in Swing.

- AncestorListener
- CaretListener
- CellEditorListener
- ChangeListener
- DocumentEvent
- DocumentListener
- HyperlinkListener
- InternalFrameListener
- ListDataListener
- ListSelectionListener
- MenuListener
- PopupMenuListener
- TableColumnModelListener
- TableModelListener
- TreeExpansionListener
- TreeModelListener
- TreeSelectionListener
- UndoableEditListener

Figure 2 shows a list of the event classes defined in the **com.sun.java.swing.event** package of **Swing 1.0.1** .

Note: Figure 2. Event classes defined in Swing.

- AncestorEvent
- CaretEvent
- ChangeEvent
- EventListenerList
- HyperlinkEvent
- InternalFrameAdapter
- InternalFrameEvent

- ListDataEvent
- ListSelectionEvent
- MenuEvent
- PopupMenuEvent
- TableColumnModelEvent
- TableModelEvent
- TreeExpansionEvent
- TreeModelEvent
- TreeSelectionEvent
- UndoableEditEvent

The two sample programs that follow illustrate the **AncestorEvent** class and the **AncestorListener** interface.

First sample program

The first sample program is named **SwingEvent10** . A complete listing of the program is shown in Listing 6 . This program illustrates the use of the **getContentPane** method to add a **JButton** object to a **JFrame** object. It also illustrates the use of an **AncestorListener** on a **JButton** . The graphic output produced by this program is shown in Figure 3 .

Note: Figure 3. Graphic screen output for SwingEvent10.



Running the program under Java 8 and physically moving the resulting **JFrame** object on the screen produced the output shown in Figure 4 .

Note: Figure 4. Text screen output from moving the JFrame.

```
Make JFrame visible
ancestorMoved method
ancestorMoved method
ancestorAdded method invoked
Event source:
javax.swing.JButton[,0,0,292x268,alignmentX=0.0,al
ignmentY=0.5,bor
der=javax.swing.plaf.BorderUIResource$CompoundBord
erUIResource@7bac7cc2,flags=29
6,maximumSize=,minimumSize=,preferredSize=,default
Icon=,disabledIcon=,disabledSe
lectedIcon=,margin=javax.swing.plaf.InsetsUIResour
ce[top=2,left=14,bottom=2,righ
```

```
t=14],paintBorder=true,paintFocus=true,pressedIcon
=,rolloverEnabled=true,rollove
rIcon=,rolloverSelectedIcon=,selectedIcon=,text=Bu
tton,defaultCapable=true]
Ancestor:
javax.swing.JFrame[frame0,0,0,300x300,layout=java.
awt.BorderLayout,tit
le=Copyright 1998,
R.G.Baldwin,resizable,normal,defaultCloseOperation
=HIDE_ON_CL
OSE,rootPane=javax.swing.JRootPane[,4,28,292x268,l
ayout=javax.swing.JRootPane$Ro
otLayout,alignmentX=0.0,alignmentY=0.0,border=,fla
gs=16777673,maximumSize=,minim
umSize=,preferredSize=],rootPaneCheckingEnabled=tr
ue]
Parent: null
Component:
javax.swing.JButton[,0,0,292x268,alignmentX=0.0,al
ignmentY=0.5,border
=javax.swing.plaf.BorderUIResource$CompoundBorderU
IResource@7bac7cc2,flags=296,m
aximumSize=,minimumSize=,preferredSize=,defaultIco
n=,disabledIcon=,disabledSelec
tedIcon=,margin=javax.swing.plaf.InsetsUIResource[
top=2,left=14,bottom=2,right=1
4],paintBorder=true,paintFocus=true,pressedIcon=,r
olloverEnabled=true,rolloverIc
on=,rolloverSelectedIcon=,selectedIcon=,text=Butto
n,defaultCapable=true]
ID value: 1
ancestorMoved method
ancestorMoved method
```

The program was originally tested using JDK 1.1.6 and Swing 1.0.1 under Windows 95. It was more recently tested using Java 8 and Windows 7.

Note that Figure 4 shows a lot more output than was produced when the same thing was done using JDK 1.1.6 and Swing 1.0.1.

Interesting code fragments for SwingEvent10

I will discuss and explain this program in fragments. As mentioned earlier, a complete listing of this program is shown in Listing 6 . Listing 1 shows the import declarations required to import the **Swing** packages. *(Note that this is different than was the case with JDK 1.1.6 and Swing 1.0.1. The old versions of the import directives are shown as comments in Listing 1 .)*

Note: Listing 1. Import directives for SwingEvent10.

```
import java.awt.*;
import java.awt.event.*;
//import com.sun.java.swing.*;//JDK 1.1 version
//import com.sun.java.swing.event.*;//JDK 1.1
version
import javax.swing.*;//JDK 1.2 version
import javax.swing.event.*;//JDK 1.2 version
```

The **main** method in the controlling class is simple, but we will include it in Listing 2 for continuity.

Note: Listing 2. The main method for SwingEvent10.

```
public class SwingEvent10 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class SwingEvent10
```

That brings us to the class named **GUI** from which we will instantiate our graphical user interface object and display it on the screen. The class named **GUI** begins in Listing 3 .

Note: Listing 3. Beginning of the class named GUI.

```
class GUI {
  public GUI(){//constructor
    //Create a new JFrame object, set size, title,
etc.
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");
    //Add window listener to terminate the program
    displayWindow.addWindowListener(new WProc1());

    //Create a JButton object
    JButton theButton = new JButton("Button");

    //Register an AncestorListener object on the
JButton
    theButton.addAncestorListener(new
MyAncestorListener());
```

```
    //Add the JButton to the JFrame using content
pane
    displayWindow.getContentPane().add(theButton);

    System.out.println("Make JFrame visible");
    displayWindow.setVisible(true);
  }//end constructor
```

As you can see in Listing 3 , the constructor for our **GUI** object is pretty straightforward.

We begin by instantiating a **Swing** object of type **JFrame** , setting its size, giving it a title, etc.

We also add a **WindowListener** to terminate the program when the user closes the **JFrame** object. *(There is an alternative and possibly better way to accomplish that in Java 8 than was the case in 1997. See the **setDefaultCloseOperation** method in the documentation for the **JFrame** class.)*

Then we instantiate a **Swing** object of type **JButton** and register an **AncestorListener** object on the button. We will discuss the class from which the listener was instantiated shortly.

After this, we add the **JButton** object to the **JFrame** object named **displayWindow** by first invoking the **getContentPane** method and then invoking the **add** method on the content pane.

Finally, we display a message and make the **JFrame** object visible. That concludes the constructor.

Now what about the **getContentPane** method. This is something that doesn't exist in the **AWT** .

In the **AWT** , we add components to, and otherwise manipulate the client area of a **Frame** object directly. However, in **Swing** , some "panes" are automatically placed in the client area of a **JFrame** object, and we add components to, and otherwise manipulate those panes instead of manipulating the client area of the **JFrame** object directly.

Rather than to try to explain this in my own words, I am simply going to provide a quotation from the JavaSoft documentation for the **JFrame** object, Swing, Version 1.0.1 in Figure 5 . Note that the wording in Figure 5 is the copyrighted property of JavaSoft.

Note:
Figure 5. Information regarding the content pane.
public class JFrame extends Frame implements WindowConstants, Accessible, RootPaneContainer
An extended version of java.awt.Frame that adds support for interposing input and painting behavior in front of the frames children (see glassPane), support for special children that are managed by a LayeredPane (see rootPane) and for Swing MenuBars.
The JFrame class is slightly incompatible with java.awt.Frame. JFrame contains a JRootPane as it's only child. The contentPane should be the parent of any children of the JFrame. This is different than java.awt.Frame, e.g. to add a child to an AWT Frame you'd write:
frame.add(child);
However using JFrame you need to add the child to the JFrame,s contentPane instead:
frame.getContentPane().add(child);
The same is true for setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane instead of the JFrame itself. The contentPane will always be non-null. Attempting to set it to null will cause the JFrame to throw an exception. The default contentPane will have a BorderLayout manager set on it.
Please see the JRootPane documentation for a complete description of the contentPane, glassPane, and layeredPane properties.

For the time being, and for this simple example, this all boils down to a requirement to insert the method call

```
getContentPane()
```

between the reference to the **JFrame** object and calls to **add** , **setLayout** , etc. For more complex programs, the ramifications could be more significant.

Our **GUI** class has two inner classes. One of those is a **WindowListener** class that is used to terminate the program when the user closes the **JFrame** object. It is so simple and so common that I'm not going to show it here. You can see it in the complete listing of the program in Listing 6 if you are interested.

The second inner class *(and these could just as well be implemented as top-level classes instead of inner classes)* is used to instantiate an **AncestorListener** object to be registered on the **JButton** object.

This is a little more interesting. The **AncestorListener** interface declares three methods, and as far as I know there is no adapter for this interface. Therefore, our class that implements the interface must define all three methods.

A brief description of each of the three methods is provided in Figure 6 .

Note: Figure 6. Methods of the AncestorListener interface.

- **ancestorAdded(AncestorEvent)** -- Called when the source or one of its ancestors is made visible either by setVisible(true) being called or by its being added to the component hierarchy.
- **ancestorMoved(AncestorEvent)** -- Called when either the source or one of its ancestors is moved.
- **ancestorRemoved(AncestorEvent)** -- Called when the source or one of its ancestors is made invisible either by setVisible(false) being called or by its being remove from the component hierarchy.

As you can see, whenever one of the methods is called, an object of type **AncestorEvent** is passed as a parameter. As usual, it is possible to invoke the methods of the **AncestorEvent** class and the classes extended by that class to learn more about the event.

We will define all three of the interface methods *(as required)* in our class definition. The code fragment in Listing 4 shows the definition of the **ancestorAdded** method.

Note: Listing 4. Definition of the AncestorAdded method.

```
  class MyAncestorListener implements
AncestorListener{
    //Define three methods declared in
AncestorListener
    // interface.

    public void ancestorAdded(AncestorEvent e){
      System.out.println("ancestorAdded method
invoked");
      System.out.println("Event source: " +
e.getSource());
      System.out.println("Ancestor: " +
e.getAncestor());
      System.out.println("Parent: " +
e.getAncestorParent());
      System.out.println("Component: " +
e.getComponent());
      System.out.println("ID value: " +
e.getID());
    }//end ancestorAdded()
```

When this method is called, it invokes five different methods of the incoming **AncestorEvent** object and displays the material returned from those methods. The output from invoking these methods was shown earlier in Figure 4 .

The final code fragment for this program in Listing 5 shows the definitions of the other two methods of the **AncestorListener** interface.

Note: Listing 5. Remaining two methods of the AncestorListener interface.

```
    public void ancestorRemoved(AncestorEvent e){
      System.out.println("ancestorRemoved
method");
    }//end ancestorRemoved()

    public void ancestorMoved(AncestorEvent e){
      System.out.println("ancestorMoved method");
    }//end ancestorMoved
  }//end class MyAncestorListener

//.....................................................
.....//
}//end class GUI definition
```

If you compile and run this program and observe the output as the program runs, you will see that the **ancestorAdded** method and the **ancestorMoved** method are both called when the **JFrame** object is made visible.

Following this, whenever the **JFrame** object is moved on the screen, the **ancestorMoved** method will be called.

Iconifying and then deiconifying the **JFrame** object will also cause the **ancestorMoved** method to be called.

Program listing for SwingEvent10

Listing 6 contains a complete listing for the program named **SwingEvent10**.

Note: Listing 6. Complete listing of program named SwingEvent10.

```
/*File SwingEvent10.java Copyright 1998,
R.G.Baldwin
Code updated 07/10/16 to be Java 8 compatible.
Rev 1/12/99 to be JDK 1.2 Swing compatible

Illustrates use of getContentPane() to add a
JButton to
a JFrame.

Illustrates use of AncestorListener on a JButton.

Running the program and moving the resulting
JFrame on the
screen produced the following output.  Note that
line breaks
were manually added to this presentation to make
the lines
fit in this format. Note that these outputs don't
seem to
provide a good match for the descriptions and
names of the
methods in the JavaSoft documentation.
```

```
Make JFrame visible
ancestorAdded method invoked
Event source: com.sun.java.swing.JButton[,0,0,0x0,

invalid,layout=com.sun.java.swing.OverlayLayout]
Ancestor: com.sun.java.swing.JButton[,0,0,0x0,

invalid,layout=com.sun.java.swing.OverlayLayout]
Parent:
com.sun.java.swing.JPanel[null.contentPane,0,0,0x0
,

invalid,layout=com.sun.java.swing.JRootPane$1]
Component: com.sun.java.swing.JButton[,0,0,0x0,

invalid,layout=com.sun.java.swing.OverlayLayout]
ID value: 1
ancestorMoved method
ancestorMoved method

Tested using JDK 1.1.6 and Swing 1.0.1 under
Win95.
**********************************************
********/
import java.awt.*;
import java.awt.event.*;
//import com.sun.java.swing.*;//JDK 1.1 version
//import com.sun.java.swing.event.*;//JDK 1.1
version
import javax.swing.*;//JDK 1.2 version
import javax.swing.event.*;//JDK 1.2 version

public class SwingEvent10 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class SwingEvent10
```

```
//=============================================
=======//

//The following class is used to instantiate a
// graphical user interface object.
class GUI {
  public GUI(){//constructor
    //Create a new JFrame object, set size, title,
etc.
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");
    //Add window listener to terminate the program
    displayWindow.addWindowListener(new WProc1());

    //Create a JButton object
    JButton theButton = new JButton("Button");

    //Register an AncestorListener object on the
JButton
    theButton.addAncestorListener(new
MyAncestorListener());

    //Add the JButton to the JFrame using content
pane
    displayWindow.getContentPane().add(theButton);

    System.out.println("Make JFrame visible");
    displayWindow.setVisible(true);
  }//end constructor

//.............................................
.....//
  //Begin inner class definitions
  //The following listener is used to terminate
the program
```

```
  // when the user closes the frame.
  class WProc1 extends WindowAdapter{
    public void windowClosing(WindowEvent e){
      System.exit(0);
    }//end windowClosing()
  }//end class WProc1

//.....
.....//

  //Define an AncestorListener class
  class MyAncestorListener implements
AncestorListener{
    //Define three methods declared in
AncestorListener
    // interface.

    public void ancestorAdded(AncestorEvent e){
      System.out.println("ancestorAdded method
invoked");
      System.out.println("Event source: " +
e.getSource());
      System.out.println("Ancestor: " +
e.getAncestor());
      System.out.println("Parent: " +
e.getAncestorParent());
      System.out.println("Component: " +
e.getComponent());
      System.out.println("ID value: " +
e.getID());
    }//end ancestorAdded()

    public void ancestorRemoved(AncestorEvent e){
      System.out.println("ancestorRemoved
method");
    }//end ancestorRemoved()
```

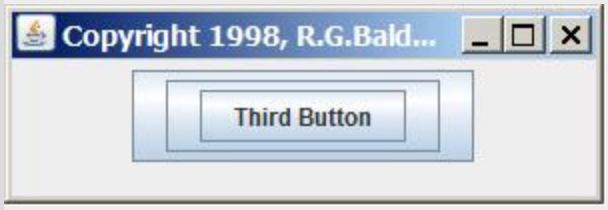
```
    public void ancestorMoved(AncestorEvent e){
      System.out.println("ancestorMoved method");
    }//end ancestorMoved
  }//end class MyAncestorListener

//.....................................................
.....//
}//end class GUI definition
//=====================================================
=======//
```

Second sample program

This program named **SwingEvent11** further illustrates the use of an **AncestorListener** on a **JButton** . The graphic screen output for this program is shown in Figure 7 .

Note: Figure 7. Graphic screen output for SwingEvent11.



More importantly, this program illustrates the very important fact that **JButton** objects are containers that can contain other objects including other **JButton** objects.

This program stacks three **JButton** objects on top of one another with the stack of three **JButton** objects being placed on a **JFrame** object.

ActionListener objects are registered on each of the buttons to trap an **actionPerformed** event when the button is clicked and to display the source of the event.

Running the program using Java 8 and carefully clicking each of the three buttons in succession from the top of the stack to the bottom of the stack, and then moving the **JFrame** object on the screen produces the output shown in Figure 8 .

Note: Figure 8. Text output from clicking buttons.

```
Make JFrame visible
In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
```

```
Event source: Third Button
In ancestorAdded method
Event source: First Button
In ancestorAdded method
Event source: Second Button
In ancestorAdded method
Event source: Third Button
In actionPerformed method
Event source: Third Button
In actionPerformed method
Event source: Second Button
In actionPerformed method
Event source: First Button
In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
```

This program was originally tested using JDK 1.1.6 and Swing 1.0.1 under Windows 95. More recently it was tested using Java 8 under Windows 7. As mentioned above, the output shown in Figure 8 was produced using Java 8.

Interesting code fragments for SwingEvent11

As before, I will discuss and explain this program using code fragments. A complete listing of the program is shown in Listing 10 .

The controlling class and the **main** method for this program is the same as the previous program, so I won't discuss it here.

The **GUI** class is shown in Listing 7 .

Note: Listing 7. The constructor for the GUI class.

```
class GUI {
  public GUI(){//constructor
    //...snip

    displayWindow.getContentPane().setLayout(
                                          new
FlowLayout());
    //...snip

    JButton firstButton = new JButton("First
Button");
    JButton secondButton = new JButton("Second
Button");
    JButton thirdButton = new JButton("Third
Button");

    //Stack the three JButton objects on top of
one
    // another.
    firstButton.add(secondButton);
    secondButton.add(thirdButton);

    //Register an AncestorListener object on each
JButton
    firstButton.addAncestorListener(
```

```
                                        new
MyAncestorListener());
    secondButton.addAncestorListener(
                                        new
MyAncestorListener());
    thirdButton.addAncestorListener(
                                        new
MyAncestorListener());

    //Register an ActionListener object on each
JButton
    firstButton.addActionListener(new
MyActionListener());
    secondButton.addActionListener(new
MyActionListener());
    thirdButton.addActionListener(new
MyActionListener());

    //...snip

  }//end constructor
```

Much of the code in the constructor for the **GUI** class is also the same as in the previous program so I deleted it from Listing 7 for brevity.

Note the use of **getContentPane** when setting the layout manager as described earlier.

Three **JButton** objects are instantiated. Then the three buttons are stacked by adding **secondButton** to **firstButton** , and by adding **thirdButton** to **secondButton** .

An **AncestorListener** object is registered on all three of the buttons and then an **ActionListener** object is registered on all three of the buttons.

The remainder of the constructor is the same as before and was deleted for brevity.

The ancestor listener class is very similar to the previous version, so the definition for only one of the methods is shown in Listing 8 .

Note: Listing 8. Definition of the ancestor listener class.

```
  class MyAncestorListener implements
AncestorListener{
    public void ancestorAdded(AncestorEvent e){
      System.out.println("In ancestorAdded
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());
    }//end ancestorAdded()

    //...snip
  }//end class MyAncestorListener
```

Note the requirement for downcasting in this version of the method. This is because invocation of the **getSource** method returns an object of type **Object** and it must be downcast to type **JButton** to be useful.

Listing 9 shows the **ActionListener** class that traps action events on the buttons when they are clicked and presents appropriate output on the screen.

Note: Listing 9. Definition of the action listener class.

```
  class MyActionListener implements
ActionListener{
    public void actionPerformed(ActionEvent e){
      System.out.println("In actionPerformed
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());
    }//end actionPerformed()
  }//end class MyActionListener
```

Program listing for SwingEvent11

Listing 10 contains a complete listing of the program named **SwingEvent11**.

Note: Listing 10. Complete listing of program named SwingEvent11.

```
/*File SwingEvent11.java Copyright 1998,
R.G.Baldwin
Code updated 7/10/16 for Java 8 compatibility.

Revised 1.12/98 for JDK 1.2 Swing compatibility.

Further illustrates use of AncestorListener on a
JButton.
Also see SwingEvent10.java.

Illustrates that JButton objects are containers
that
can contain other JButton objects.
```

```
This program stacks three JButton objects on top
of one
another with the stack of three JButton objects
being
placed on a JFrame object.

Running the program and carefully clicking each of
the
three buttons in succession from the top of the
stack to
the bottom of the stack, and then moving the
JFrame object
on the screen produces the following output. Note
that
some blank lines were manually inserted to make it
easier
to follow this material.

Make JFrame visible
In ancestorAdded method
Event source: First Button
In ancestorAdded method
Event source: Second Button
In ancestorAdded method
Event source: Third Button

In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button

In ancestorMoved method
Event source: First Button
In ancestorMoved method
```

```
Event source: Second Button
In ancestorMoved method
Event source: Third Button

In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button
In ancestorMoved method
Event source: Third Button

In actionPerformed method
Event source: Third Button
In actionPerformed method
Event source: Second Button
In actionPerformed method
Event source: First Button

In ancestorMoved method
Event source: First Button
In ancestorMoved method
Event source: Second Button
In ancestorMoved method
Event source: Third Button



Tested using JDK 1.1.6 and Swing 1.0.1 under
Win95.
**********************************************
********/
import java.awt.*;
import java.awt.event.*;
//import com.sun.java.swing.*;//JDK 1.1 version
//import com.sun.java.swing.event.*;//JDK 1.1
version
import javax.swing.*;//JDK 1.2 version
import javax.swing.event.*;//JDK 1.2 version
```

```
public class SwingEvent11 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class SwingEvent11
//===========================================
=======//

//The following class is used to instantiate a
// graphical user interface object.
class GUI {
  public GUI(){//constructor
    //Create a new JFrame object, set size, title,
etc.
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,100);
    displayWindow.setTitle("Copyright 1998,
R.G.Baldwin");

    //Note required use of getContentPane() in
following
    // statement.
    displayWindow.getContentPane().setLayout(
                                          new
FlowLayout());
    //Add window listener to terminate the program
    displayWindow.addWindowListener(new WProc1());

    //Create three JButton objects
    JButton firstButton = new JButton("First
Button");
    JButton secondButton = new JButton("Second
Button");
    JButton thirdButton = new JButton("Third
Button");
```

```
    //Stack the three JButton objects on top of
one
    // another.
    firstButton.add(secondButton);
    secondButton.add(thirdButton);

    //Register an AncestorListener object on each
JButton
    firstButton.addAncestorListener(
                                    new
MyAncestorListener());
    secondButton.addAncestorListener(
                                    new
MyAncestorListener());
    thirdButton.addAncestorListener(
                                    new
MyAncestorListener());

    //Register an ActionListener object on each
JButton
    firstButton.addActionListener(new
MyActionListener());
    secondButton.addActionListener(new
MyActionListener());
    thirdButton.addActionListener(new
MyActionListener());

    //Add the JButton to the JFrame using content
pane

displayWindow.getContentPane().add(firstButton);

    System.out.println("Make JFrame visible");
    displayWindow.setVisible(true);
  }//end constructor

//.............................................
```

```
.....//
  //Begin inner class definitions

  //The following listener is used to terminate
the
  // program when the user closes the frame.
  class WProc1 extends WindowAdapter{
    public void windowClosing(WindowEvent e){
      System.exit(0);
    }//end windowClosing()
  }//end class WProc1

//.....
.....//

  //Define an AncestorListener class
  class MyAncestorListener implements
AncestorListener{
    //Define three methods declared in
AncestorListener
    // interface.  Note the required downcasting.

    public void ancestorAdded(AncestorEvent e){
      System.out.println("In ancestorAdded
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());
    }//end ancestorAdded()

    public void ancestorRemoved(AncestorEvent e){
      System.out.println("In ancestorRemoved
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());
    }//end ancestorRemoved()
```

```
    public void ancestorMoved(AncestorEvent e){
      System.out.println("In ancestorMoved
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());

    }//end ancestorMoved
  }//end class MyAncestorListener

//.....................................................
.....//

  //Define an ActionListener class
  class MyActionListener implements
ActionListener{
    public void actionPerformed(ActionEvent e){
      System.out.println("In actionPerformed
method");
      System.out.println("Event source: " +

((JButton)e.getSource()).getActionCommand());
    }//end actionPerformed()
  }//end class MyActionListener

//.....................................................
.....//

}//end class GUI definition
//=====================================================
=======//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java085 Swing, New Event Types in Swing
- File: Java085.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java087 Swing, Understanding getContentPane() and other JFrame Layers
The purpose of this lesson is to help you understand why you need to use getContentPane() or some similar method call to add components to a JFrame object, remove components from a JFrame object, or set the layout for a JFrame object. Note that this also applies to container objects of type JInternalFrame and JDialog as well.

Revised: Sun Jul 10 12:16:18 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- General discussion
    - Synopsis
    - More detailed discussion
- Sample program
    - Interesting code fragments
    - Complete program listing
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains how to write programs that handle events in Java.

Students in Prof. Baldwin's **ITSE 2317 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Graphic screen output for the program named SwingPane01.
- Figure 2. Screen output after clicking green button.
- Figure 3. Text output.

Listings

- Listing 1. Import declarations.
- Listing 2. Beginning of the class named SwingPane01.
- Listing 3. Beginning of the constructor.
- Listing 4. Get and display types of panes.
- Listing 5. Instantiate JLabel and JTextField objects.
- Listing 6. Instantiate a green JButton object.
- Listing 7. Instantiate a yellow JButton object.
- Listing 8. The end of the constructor.
- Listing 9. The class named MyActionListener.
- Listing 10. Complete listing of SwingPane01.

Introduction

Previous lessons involving Swing told you that when you add a component to a **JFrame** object, unlike with the **AWT** , you must use statements similar to the following.

Note:

```
myJFrameObject.getContentPane().add(myChildCompone
nt);
```

As you have probably suspected, there is a lot more to it than just inserting **getContentPane()** between the object reference and the **add** method.

The purpose of this lesson is to help you understand why you need to use **getContentPane** or some similar method call to add components to a **JFrame** object, remove components from a **JFrame** object, or set the layout for a **JFrame** object.

Note that this also applies to container objects of type **JInternalFrame** and **JDialog** as well.

General discussion

When programming with the **AWT** , you can place a **Panel** object in a **Frame** object and place other components on the **Panel** object if you wish, or you can simply place other components directly on the viewable area of the **Frame** object.

Note:

The viewable area of **Frame** or a **JFrame** is the *bounds* minus the *insets* . Insets are used to account for the space covered by the borders and the values of the insets on all four sides are available by invoking the method named **getInsets()** on the object.

However, you cannot place components directly on the viewable area of a **JFrame** object.

An object of type **JRootPane** is automatically placed in the **JFrame** completely covering the viewable area of the **JFrame** object.

One way or another, if you want to place components in the **JFrame** , you must deal with the **JRootPane** object.

One way to deal with it, and this is the way that you will find recommended in many current discussions on the subject, is simply ignore the technical details and insert a call to the **getContentPane** method between the reference to the **JFrame** and the **add** method.

My preference is to first understand why you need to do that, and in those cases where that is the appropriate thing to do, go ahead and do it.

In those cases where invoking **getContentPane** is not the most appropriate thing to do, do the thing that is most appropriate.

Note: You will find much discussion in this lesson about some components being on top of other components. These discussions apply only to Swing lightweight components. Remember that heavyweight *(AWT)* components are always on top of lightweight *(Swing)* components regardless of what the text in this lesson may say.

Synopsis

JRootPane is a container. The instance of **JRootPane** that is automatically placed in a **JFrame** object contains at least two other objects.

- An object of type **JLayeredPane** that we will refer to simply as the ***layeredPane*** .
- An object of type **JPanel** that we will refer to simply as the ***glassPane*** .

Note:
There is also an optional *menuBar* object that comes in here somewhere, but since it is not essential for understanding what is going on at this level, I have elected to defer the discussion until the discussion on Swing menus.

The *layeredPane* and the *glassPane* also fill the viewable area of the **JFrame** object. The *glassPane* is on top of the *layeredPane* , and as the name might imply, it is *normally* transparent.

Components on the *layeredPane* are visible through the *glassPane* . Mouse events are capable of penetrating the *glassPane* and reaching components on the *layeredPane* as long as the *glassPane* is transparent.

The *layeredPane* contains another component of type **JPanel** that we will refer to as the ***contentPane*** . This is the same pane that we are dealing with when we invoke the **getContentPane** method on the **JFrame** object.

The *layeredPane* has a very interesting behavior.

In the same sense that we can add components to the *contentPane* , we can also add components to the *layeredPane* .

When we add components to the *contentPane* , they are all added at the same layer and components added first are painted on top of components added later.

However, when we add components to the *layeredPane* , we can specify the layer number that we want the component to be drawn on.

Components drawn on layers with high *(algebraically signed)* numbers are painted on top of components on layers with smaller *(algebraically signed)* numbers.

Note:
Large negative numbers are smaller than small positive numbers in this case with the smallest allowable layer number being -29999.

In addition, a variety of methods are available to move components from one layer to another at runtime. This gives us the ability to not only control the order in which components are painted initially, but also to modify that order later at runtime.

This layering concept is a much more powerful approach than simply inserting **getContentPane** between the reference to the **JFrame** object and the invocation of the **add** method.

Note:
Layering has a long history of beneficial results in computer graphics such as Computer Aided Design and game programming.

So, where does the *contentPane* object that belongs to the *layeredPane* fit into this?

Components on the *contentPane* are painted behind all components that may be added directly to the *layeredPane* . In fact, the layer position for the *contentPane* is effectively layer number -30000. The smallest layer number that can be used to place a component directly on a layer is -29999. *(We will place one there in our sample program later.)*

More detailed discussion

As mentioned earlier, a **JRootPane** contains a *glassPane (**JPanel**)* and a *layeredPane (**JLayeredPane**)*. The *layeredPane* contains an optional *menuBar (**JMenuBar**)* and a *contentPane (**JPanel**)*. The *menuBar* component is optional and may or may exist at any given time. The *layeredPane* , *contentPane* , and *glassPane* will always be available.

The syntax for dealing with these panes is somewhat different from what you are accustomed to. For example, the typical syntax for adding a component to a container would look something like the following:

Note:

```
parentContainer.add(childObject);
```

However, the required syntax for dealing with these panes is more like one of the following:

Note:

```
myJFrameObject.getContentPane().add(myChildCompone
nt);

myJFrameObject.getLayeredPane().add(
                    myChildComponent, new
Integer(5));
```

The first statement above adds a component to the *contentPane* . The second statement adds a component to layer number 5 of the *layeredPane* .

The same concepts apply when removing components, setting layout managers, etc.

The *contentPane* has a **BorderLayout** manager by default. The *layeredPane* has no layout manager *(null)* by default as you will see in the sample program that follows later in this lesson.

If a **JMenuBar** component is set on the **JRootPane** , it is positioned along the upper edge of the frame. The *contentPane* is adjusted in location and size to fill the remaining area.

If you examine the documentation for **JLayeredPane** , you will see that the class has the following fields which are all *public static final* **Integer** symbolic constants.

Note:

- **DEFAULT_LAYER** - Object defining the Default layer. Equivalent to new Integer(0).
- **PALETTE_LAYER** - Object defining the Palette layer. Equivalent to new Integer(100).
- **MODAL_LAYER** - Object defining the Modal layer. Equivalent to new Integer(200).
- **POPUP_LAYER** - Object defining the Popup layer. Equivalent to new Integer(300).
- **DRAG_LAYER** - Object defining the Drag layer. Equivalent to new Integer(400).
- **FRAME_CONTENT_LAYER** - Object defining the Frame Content layer. This layer is normally only used to position the **contentPane** and menuBar components of **JFrame** . Equivalent to new Integer(-30000).

The **JMenuBar** and the *contentPane* are added to the *layeredPane* component at the following layer (-30000):

```
JLayeredPane.FRAME_CONTENT_LAYER
```

As mentioned earlier, the *layeredPane* object is an instance of the **JLayeredPane** class. The purpose of this object is to be the parent of all children of the **JRootPane** .

Also, as mentioned earlier, this object provides the ability to add components at several layers. This is very useful when working with popup menus, dialog boxes, during dragging, or for any other situation in which you might want to separate graphic objects onto different layers for display purposes.

Also as indicated earlier, the *glassPane* is always added as the first child of the **JRootPane** . This causes the **glassPane** to always be on the top of the stack. By default, the **glassPane** is not visible and is transparent. Thus, mouse events can normally penetrate the *glassPane* and impinge upon the components below it.

It is theoretically possible to draw components on the *glassPane (although I haven't been able to purposely do it as of 5/12/98).* When this is done, those components shadow the components below them preventing mouse events from impinging on the components down below.

Components drawn on the *glassPane* will always be above all other lightweight components in the stacking order of components *(but not over top of heavyweight components).* Thus, the *glassPane* can be used to assure that such components as popup menus and tool tips are always on top of the other components on the screen.

As mentioned earlier, by default, the *glassPane* is not visible. Developers should use setVisible on the *glassPane* to control when the *glassPane* displays over the other children.

The layout manager used by **JRootPane** insures that the following is true:

Note:

- The **glassPane** , if present, fills the entire viewable area of the **JRootPane** (bounds - insets).
- The **layeredPane** fills the entire viewable area of the **JRootPane** . (bounds - insets)
- The **menuBar** is positioned at the upper edge of the **layeredPane** ().
- The **contentPane** fills the entire viewable area, minus the **MenuBar** , if present.

If you replace the layout manager of the **JRootPane** with a different layout manager, you are responsible for managing all of those views.

So now we know that the **JRootPane** object contains an object of type **JLayeredPane** . While **JLayeredPane** manages it's list of children like a **Container** , it also allows for the definition of several layers within itself.

Children in the same layer are managed exactly like the normal **Container** object. However, children in higher layers display above the children in lower layers. Each layer has a distinct integer number.

Apparently each Swing **Component** has a layer attribute *(but I haven't been able to find out anything about it other than as described below).*

There are at least three ways to set the layer attribute on a **Component** :

- Passing an Integer object with a literal integer value during the add call: **layeredPane.add(child,new Integer(10));**
- Passing one of the symbolic constants mentioned earlier during the add call: **layeredPane.add(child,JLayeredPane.DEFAULT_LAYER)**
- Calling the following set method on the **JLayeredPane** that will be the parent of the component and passing the name of the component and an **int** that specifies the layer number: **layeredPaneParent.setLayer(child,10)**

In this third case, the layer should be set before adding the child to the parent.

Layers with higher numbers display on top of layers with lower numbers.

Higher and lower in this case includes the algebraic sign of the layer. For example, a large negative number is a lower layer than a small positive number.

These layers are simply a logical construct and layout managers will affect all child components without regard for layer settings. Therefore, to take advantage of this layering capability, you may need to design your own layout manager or use absolute layout *(null layout manager)* .

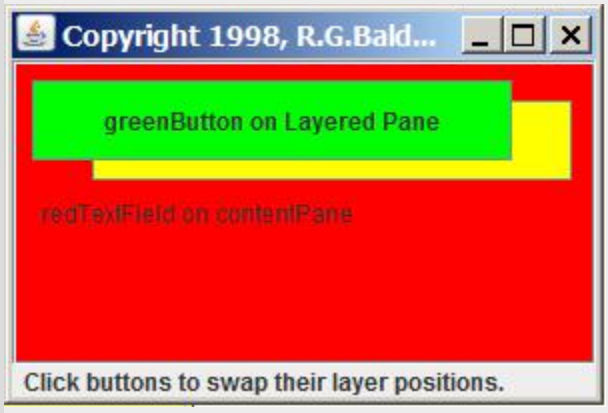
We will see some examples in the sample program that follows.

Sample program

The primary purpose of this program is to illustrate the use of both the *contentPane* and the *layeredPane* .

The program places a **JFrame** object on the screen as the primary GUI as shown in Figure 1 .

Note: Figure 1. Graphic screen output for the program named SwingPane01.


Copyright 1998, R.G.Bald...
greenButton on Layered Pane
redTextField on contentPane
Click buttons to swap their layer positions.

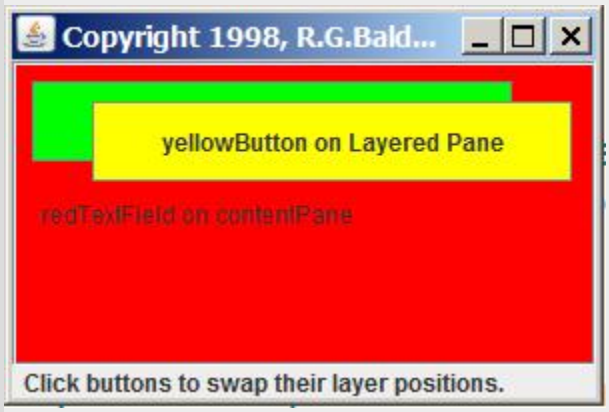
A red **JTextField** object and a **JLabel** object are placed on the *contentPane* on the **JFrame** object. The **JTextField** object is placed in the *Center* of the **JFrame** using the default border layout. The **JLabel** is placed in the *South* position on the **JFrame** object.

A green **JButton** object and a yellow **JButton** object are placed on the *layeredPane* . Apparently the default layout for the *layeredPane* is absolute or null. The **JButton** objects are purposely placed so as to partially overlap. Both buttons appear on top of the red **JTextField** object on the *contentPane* . *(Components on layered panes are always on top of components on the* ***contentPane*** *).*

The green **JButton** is initially placed at layer position +1 on the layered pane and the yellow **JButton** is initially placed at layer position -29999 which is the limit in the negative direction.

Action listeners are registered on the two buttons. When the top *(green)* button is clicked, the action is to swap the layer positions of the two buttons causing the yellow button to move to the top as shown in Figure 2 .

Note: Figure 2. Screen output after clicking green button.


Copyright 1998, R.G.Bald...
yellowButton on Layered Pane
redTextField on contentPane
Click buttons to swap their layer positions.

The program was originally tested using JDK 1.1.6 and Swing 1.0.1 under Windows 95. More recently it was tested using Java 8 under Windows 7.

Several lines of code are included in the program to investigate the parent child relationships of the various panes. The text output from this part of the program, when run using Java 8, is shown in Figure 3 .

Note: Figure 3. Text output.

```
Root pane is: class javax.swing.JRootPane
Parent of root pane is
SwingPane01[frame0,0,0,0x0,invalid,hidden,layout=j
ava.awt
.BorderLayout,title=,resizable,normal,defaultClose
Operation=HIDE_ON_CLOSE,rootPa
ne=javax.swing.JRootPane[,0,0,0x0,invalid,layout=j
avax.swing.JRootPane$RootLayou
t,alignmentX=0.0,alignmentY=0.0,border=,flags=1677
7673,maximumSize=,minimumSize=
,preferredSize=],rootPaneCheckingEnabled=true]

Glass pane is: class javax.swing.JPanel
Parent of glass pane is
javax.swing.JRootPane[,0,0,0x0,invalid,layout=java
x.swin
g.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0
.0,border=,flags=16777673,maxi
mumSize=,minimumSize=,preferredSize=]

Layered pane is: class javax.swing.JLayeredPane
Parent of layered pane is
javax.swing.JRootPane[,0,0,0x0,invalid,layout=java
x.sw
ing.JRootPane$RootLayout,alignmentX=0.0,alignmentY
=0.0,border=,flags=16777673,ma
```

```
ximumSize=,minimumSize=,preferredSize=]

Content pane is: class javax.swing.JPanel
Parent of content pane is
javax.swing.JLayeredPane[null.layeredPane,0,0,0x0,
inva
lid,alignmentX=0.0,alignmentY=0.0,border=,flags=0,
maximumSize=,minimumSize=,pref
erredSize=,optimizedDrawingPossible=true]
```

The JavaSoft documentation indicates that the *contentPane* is placed at an equivalent layer position of -30000 in the *layeredPane* .

The *glassPane* is something of a mystery to me at this point in time (5/12/98). I was unable to draw on the *glassPane* . An attempt to place either a **JButton** object or a **JToolTip** object on the *glassPane* was rejected by the compiler with error messages that the **JButton** and the **JToolTip** could not be converted to type **JPopupMenu** . I don't know if this is proper behavior or a bug.

Interesting code fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 10 . Listing 1 shows the import declarations required to import the Swing and AWT packages using Java 8. *(The older import declaration is shown as a comment.)*

Note: Listing 1. Import declarations.

```
import java.awt.*;
import java.awt.event.*;
```

```
//import com.sun.java.swing.*;//JDK 1.1 version
import javax.swing.*;//JDK 1.2 version
```

The class definition for **SwingPane01** begins in Listing 2 .

Note: Listing 2. Beginning of the class named SwingPane01.

```
class SwingPane01 extends JFrame{
  JLayeredPane theLayeredPane;
  JButton greenButton;
  JButton yellowButton;

  public static void main(String[] args){
    new SwingPane01();
  }//end main
```

As you can see in Listing 2 , the controlling class in this program is an object of type **JFrame** because it extends **JFrame** . This fragment also declares some instance variables that are required later in the **ActionEvent** handler to swap the two buttons between layers.

The **main** method is also included in this fragment simply to provide continuity.

Listing 3 shows the beginning of the constructor where much of the work in this program is accomplished.

Note: Listing 3. Beginning of the constructor.

```
  SwingPane01(){//constructor
    //Get a ref to the layered pane for later use.
    theLayeredPane = this.getLayeredPane();
    int frameWidth = 300;
    int frameHeight = 200;
```

Listing 3 begins by getting a reference to the *layeredPane* to make it more convenient to work with it later. The width and height values for the **JFrame** object are set in this fragment as well.

Listing 4 shows the code that is used to discover and display the types of the different panes along with the parent-child relationships in the hierarchy of panes. This is the code that produced most of the text output shown in Figure 3 .

Note: Listing 4. Get and display types of panes.

```
    System.out.println("Root pane is: " +

this.getRootPane().getClass());
    System.out.println("Parent of root pane is " +
                    this.getRootPane().getParent()
+ "\n");

    System.out.println("Glass pane is: " +

this.getGlassPane().getClass());
    System.out.println("Parent of glass pane is "
+

                    this.getGlassPane().getParent()
+ "\n");
```

```
    System.out.println("Layered pane is: " +
this.getLayeredPane().getClass());
    System.out.println("Parent of layered pane is
" +
                   this.getLayeredPane().getParent()
+ "\n");

    System.out.println("Content pane is: " +
this.getContentPane().getClass());
    System.out.println("Parent of content pane is
" +
                   this.getContentPane().getParent()
+ "\n");
```

This code is all pretty intuitive. You should be able to surmise what it is doing simply from the names of the methods being invoked.

Listing 5 instantiates a **JLabel** object and a **JTextField** object and places them on the *contentPane* .

Note: Listing 5. Instantiate JLabel and JTextField objects.

```
    JLabel theLabel = new JLabel(
         "  Click buttons to swap their layer
positions.");
    this.getContentPane().add(theLabel,"South");

    //Put a red JTextField in the Center of the
```

```
JFrame on
    // the content pane.
    JTextField redTextField = new JTextField(
                          "    redTextField on
contentPane");
    redTextField.setBackground(Color.red);

this.getContentPane().add(redTextField,"Center");
```

The default layout manager for the *contentPane* is **BorderLayout** . This code places the red **JTextField** in the **Center** position of the *contentPane* and places the **JLabel** in the **South** position.

For the *metal* Look and Feel, the **JLabel** appears to be part of the border at the bottom of the **JFrame** object as shown in Figure 2 .

The *South* **JLabel** provides instructions regarding the use of the **JButton** objects that will also be placed on the **JFrame** object.

Listing 6 instantiates a green **JButton** object and places it on the *layeredPane* at layer number one (1).

Note: Listing 6. Instantiate a green JButton object.

```
    greenButton = new JButton(
                              "greenButton on
Layered Pane");
    greenButton.setBackground(Color.green);
    greenButton.setBounds(10,10,240,40);
    greenButton.addActionListener(new
MyActionListener());
```

```
    theLayeredPane.add(greenButton,new
Integer(1));
```

Note that the **setBounds** method is used to establish the location and size of the button in absolute pixel coordinates. Recall that I said earlier that apparently the default layout manager for the *layeredPane* is null. Otherwise, it would have been necessary for me to set it to null before placing this object on an absolute location and size basis.

Also note that an **ActionListener** object is registered on the **JButton** object. We will see the class that defines the behavior of the listener object later.

Listing 7 instantiates a yellow **JButton** object and places it on the *layeredPane* at layer number -29999. This is the bottom-most layer of all programmable layers in the *layeredPane* . The **contentPane** falls immediately below this layer.

Note: Listing 7. Instantiate a yellow JButton object.

```
    yellowButton = new JButton(
                            "yellowButton on
Layered Pane");
    yellowButton.setBackground(Color.yellow);
    yellowButton.setBounds(40,20,240,40);
    yellowButton.addActionListener(new
MyActionListener());
    theLayeredPane.add(yellowButton,new
Integer(-29999));
```

Listing 8 sets the title, size, visibility, etc., of the **JFrame** object and register an anonymous listener to terminate the program when the user closes the **JFrame** .

Note: Listing 8. The end of the constructor.

```
    this.setTitle("Copyright 1998, R.G.Baldwin");
    this.setSize(frameWidth,frameHeight);
    this.setVisible(true);

//================================================
===//
    //Anonymous inner class to terminate program.
    this.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);}});//end
addWindowListener

  }//end constructor
```

Listing 8 also signals the end of the constructor.

Listing 9 shows the **ActionListener** class from which listener objects are registered on the two **JButton** objects.

Note: Listing 9. The class named MyActionListener.

```
  class MyActionListener implements
ActionListener{
```

```
    public void actionPerformed(ActionEvent e){
      if(e.getActionCommand().equals(
                             "greenButton on Layered
Pane")){

theLayeredPane.setLayer(greenButton,-29999);
        theLayeredPane.setLayer(yellowButton,1);
      }else{
        theLayeredPane.setLayer(greenButton,1);

theLayeredPane.setLayer(yellowButton,-29999);
      }//end else
    }//end actionPerformed()
  }//end class MyActionListener
```

This is a standard listener class that defines the **actionPerformed** method.

The behavior of the **actionPerformed** method in this case is to cause the layer positions of the two **JButton** objects to be swapped whenever the button on top is clicked. This causes the **JButton** object immediately below the one on the top to move to the top of the stack.

Note that this method invokes the **setLayer** method of the **JLayeredPane** class to rearrange the positions of the two **JButton** objects at runtime.

Complete program listing

Listing 10 shows a complete listing of the program named **SwingPane01** .

Note: Listing 10. Complete listing of SwingPane01.

```
/*File SwingPane01 Copyright 1998, R.G.Baldwin
Updated 7/10/16 for Java 8 compatibility.

Revised 1/12/99 for JDK 1.2 Swing compatibility

The purpose of this program is to illustrate the
use of
the content pane and the layered pane.

The program places a JFrame object on the screen
as the
primary GUI.

A red JTextField and a white JLabel are placed on
the
content pane on the JFrame object.  (The white
label is
actually rendered as gray in the metal L&F.)

The JTextField is placed in the Center of the
JFrame using
the default border layout.  The JLabel is placed
in the
South position on the JFrame object.

A green JButton and a yellow JButton are placed on
the
layered pane of the JFrame object.  Apparently the
default
layout for the layered pane is absolute or null.
The
JButton objects are purposely placed so as to
partially
overlap.  Both buttons appear on top of the
JTextField

object on the content pane.
```

```
The green JButton is initially placed at layer
position +1
on the layered pane and the yellow JButton is
initially
placed at layer position -29999 which is the limit
in the
negative direction.

Action listeners are registered on the two
buttons.  When
the top button is clicked, the action is to swap
the layer
positions of the two buttons causing the other one
to
move to the top layer position.

Several lines of code are included to investigate
the
parent child relationships of the various panes.
The
output from this part of the program follows.
Note that
line breaks were manually inserted here to force
the
material to fit in this format.


Root pane is: class com.sun.java.swing.JRootPane
Parent of root pane is SwingPane01
     [frame0,0,0,0x0,invalid,hidden,

layout=java.awt.BorderLayout,resizable,title=]

Glass pane is: class com.sun.java.swing.JPanel
Parent of glass pane is

com.sun.java.swing.JRootPane
     [,0,0,0x0,invalid,
```

```
layout=com.sun.java.swing.JRootPane$RootLayout]

Layered pane is: class
com.sun.java.swing.JLayeredPane
Parent of layered pane is
com.sun.java.swing.JRootPane
     [,0,0,0x0,invalid,

layout=com.sun.java.swing.JRootPane$RootLayout]

Content pane is: class com.sun.java.swing.JPanel
Parent of content pane is
com.sun.java.swing.JLayeredPane
     [null.layeredPane,0,0,0x0,invalid]

An interpretation of the above is:
  The root pane is a child of the JFrame object.
  The glass page is a child of the root pane.
  The layered pane is a child of the root pane.
  The content pane is a child of the layered pane.

  The root pane is of type JRootPane.
  The glass pane is of type JPanel.
  The layered pane is of type JLayeredPane.
  The content pane is of type JPanel.

The JavaSoft documentation indicates that the
content pane
is placed at layer position -30000 in the layered
pane.

The glass pane is something of a mystery at this
point.
I was unable to draw upon the glass pane.  An

attempt to
place either a JButton object or a JToolTip object
```

```
on the
glass plane was rejected by the compiler with
error
messages that the JButton and the JToolTip could
not be
converted to type JPopupMenu.

Tested using JDK 1.1.6 and Swing 1.0.1 under
Win95.
**********************************************
********/

import java.awt.*;
import java.awt.event.*;
//import com.sun.java.swing.*;//JDK 1.1 version
import javax.swing.*;//JDK 1.2 version

class SwingPane01 extends JFrame{//subclass JFrame
  JLayeredPane theLayeredPane;
  JButton greenButton;
  JButton yellowButton;

  public static void main(String[] args){
    new SwingPane01();
  }//end main
  //-----------------------------------------
-------//

  SwingPane01(){//constructor
    //Get a ref to the layered pane for later use.
    theLayeredPane = this.getLayeredPane();
    int frameWidth = 300;
    int frameHeight = 200;

    //Get and display types of different panes
along with
    // parent-child hierarchy.
```

```
    System.out.println("Root pane is: " +
                       this.getRootPane().getClass());
    System.out.println("Parent of root pane is " +
                       this.getRootPane().getParent()
+ "\n");

    System.out.println("Glass pane is: " +
                       this.getGlassPane().getClass());
    System.out.println("Parent of glass pane is "
+
                       this.getGlassPane().getParent()
+ "\n");

    System.out.println("Layered pane is: " +
                       this.getLayeredPane().getClass());
    System.out.println("Parent of layered pane is
" +
                       this.getLayeredPane().getParent()
+ "\n");

    System.out.println("Content pane is: " +
                       this.getContentPane().getClass());
    System.out.println("Parent of content pane is
" +
                       this.getContentPane().getParent()
+ "\n");


    //Put instructions in a JLabel on the content
pane.
    JLabel theLabel = new JLabel(
          "  Click buttons to swap their layer
positions.");
```

```
    this.getContentPane().add(theLabel,"South");

    //Put a red JTextField in the Center of the
JFrame on
    // the content pane.
    JTextField redTextField = new JTextField(
                         "    redTextField on
contentPane");
    redTextField.setBackground(Color.red);

this.getContentPane().add(redTextField,"Center");

    //Put a green JButton on the layered pane at a
layer
    // position of +1.
    greenButton = new JButton(
                              "greenButton on
Layered Pane");
    greenButton.setBackground(Color.green);
    greenButton.setBounds(10,10,240,40);
    greenButton.addActionListener(new
MyActionListener());
    theLayeredPane.add(greenButton,new
Integer(1));

    //Put a yellow JButton on the layered pane at
a layer
    // position of -29999.
    yellowButton = new JButton(
                             "yellowButton on
Layered Pane");
    yellowButton.setBackground(Color.yellow);
    yellowButton.setBounds(40,20,240,40);
    yellowButton.addActionListener(new
MyActionListener());

    theLayeredPane.add(yellowButton,new
Integer(-29999));
```

```
    //Set title, size, and visibility of JFrame
object.
    this.setTitle("Copyright 1998, R.G.Baldwin");
    this.setSize(frameWidth,frameHeight);
    this.setVisible(true);

//================================================
===//
    //Anonymous inner class to terminate program.
    this.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);}});//end
addWindowListener

  }//end constructor

//================================================
=====//

  //Inner class for listener objects which swap
the layer
  // positions of the two JButton objects when the
one on
  // the top is clicked.
  class MyActionListener implements
ActionListener{
    public void actionPerformed(ActionEvent e){
      if(e.getActionCommand().equals(
                            "greenButton on Layered
Pane")){

theLayeredPane.setLayer(greenButton,-29999);
        theLayeredPane.setLayer(yellowButton,1);
      }else{

        theLayeredPane.setLayer(greenButton,1);
```

```
theLayeredPane.setLayer(yellowButton,-29999);
      }//end else
    }//end actionPerformed()
  }//end class MyActionListener

}//end class SwingPane01
//===================================================
=======//
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java087 Swing, Understanding getContentPane() and other JFrame Layers
- File: Java087.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java090 Scrollbar (Adjustment) Event Handling
Event handling on a Scrollbar object differs from the event handling explained in previous lessons in several respects. This lesson explains those differences.

Revised: Mon Jul 11 13:04:48 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Table of contents
- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- Overview
- The sample program
    - Discussion
    - Interesting code fragments
    - Complete program listing
- Review
- Miscellaneous

Preface

This is a page from the Event Handling section of the book titled ITSE2317 - Java Programming (Intermediate) . The **Event Handling** section explains

how to write programs that handle events in Java.

Students in Prof. Baldwin's **ITSE 2317 Intermediate Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Note:
Note: An object of the **JSlider** class in Java 8 can be used for some *(but no all)* of the purposes that a **Scrollbar** object may have been used for in the early days of Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. Graphic screen output for Event18.
- Figure 2. Output produced by clicking right scrollbar button.
- Figure 3. Graphic screen output from SampProg125.java.

Listings

- Listing 1. The controlling class for Event18.java.
- Listing 2. Beginning of the GUI class.
- Listing 3. Beginning of the constructor for the GUI class.
- Listing 4. Set the unit and block increments.
- Listing 5. Complete the GUI.
- Listing 6. Beginning of class that implements AdjustmentListener.

- Listing 7. Beginning of the adjustmentValueChanged method.
- Listing 8. Get and display the value of the scrollbar.
- Listing 9. Complete listing of the program named Event18.java.
- Listing 10. Complete listing of the program named SampProg125.java.

Introduction

JDK 1.1 was formally released on February 18, 1997. This lesson was originally written on February 21, 1997 using the software in JDK 1.1. It was upgraded to JDK 1.2 on 12/18/98. It was converted to CNXML with minor updates for publication on OpenStax on 07/11/16.

Overview

Event handling on a **Scrollbar** object differs from the event handling in previous lessons in several respects. The following paragraphs contrast the **Scrollbar** with the **Mouse** in order to illustrate some of the differences.

To create a listener object for the mouse, you either implement the **MouseListener** interface, or you extend the **MouseAdapter** class.

However, to create a listener object for a **Scrollbar** , you do not implement a *ScrollbarListener* interface or extend a *ScrollbarAdapter* class, because they do not exist.

Instead, you implement the **AdjustmentListener** interface. The single method declared in the **AdjustmentListener** interface receives an **AdjustmentEvent** object as a parameter. This is one of the *semantic* events *(as opposed to low-level events)* similar to **ActionEvent** , **ItemEvent** , and **TextEvent.**

As mentioned above, there is no *AdjustmentAdapter* class. It isn't needed because the **AdjustmentListener** interface declares only one method, which is shown below:

Note:

```
public abstract void
adjustmentValueChanged(AdjustmentEvent e)
```

The **adjustmentValueChanged** method is invoked when the value of the adjustable object *(the Scrollbar object in this lesson)* has changed.

Recall also that there are five different types of mouse events declared in the **MouseListener** interface:

- mouseClicked,
- mouseEntered,
- mouseExited,
- mousePressed
- mouseReleased.

There are two additional types of mouse events that are declared in the **MouseMotionListener** interface:

- mouseDragged
- mouseMoved

Each of these different types of mouse events is represented by a method declaration in one or the other of the two interfaces defined for creating listener classes for mouse activity. You *override the method(s)* for those types of events which interest you.

There are five different types of adjustment events that correspond to the five ways of moving the bubble in a **Scrollbar** object. However, as mentioned above, there is only one method declared in the **AdjustmentListener** interface. It is named **adjustmentValueChanged** . It receives an object of type **AdjustmentEvent** as a parameter when invoked.

The type of event is encoded into the object that is passed in as a parameter along with other information such as the *value* of the **Scrollbar** and the *name* of the **Scrollbar** object. The *value* of the **Scrollbar** is determined by the position of the bubble on the **Scrollbar** .

The **AdjustmentEvent** class defines several methods that can be used to extract information from the object in case you need access to that information. This is discussed in more detail later.

The types of adjustment events *(corresponding to physical methods of moving the bubble on the **Scrollbar***) are defined as static variables in the **AdjustmentEvent** class. *(This is similar to the approach used in the earlier event model in JDK 1.0.2.)* The five different types of adjustment events and their relationship to moving the bubble are:

- **UNIT_INCREMENT** - click button on one end of the Scrollbar
- **UNIT_DECREMENT** - click button on the other end of the Scrollbar
- **BLOCK_INCREMENT** - click in space between bubble and one button
- **BLOCK_DECREMENT** - click in space between bubble and other button
- **TRACK** - drag the bubble

As indicated, the two *UNIT* types are generated by clicking on the buttons at either end of the Scrollbar.

The two *BLOCK* types are generated by clicking interior to the **Scrollbar** on either side of the bubble.

The *TRACK* event is generated by dragging the bubble.

Methods are available for setting various parameters of the **Scrollbar** including the range, the width of the bubble *(alternately referred to in the documentation as page size or visible*), the size of the unit and block increments or decrements, etc.

One might surmise that this rather general purpose adjustment approach was defined to support a family of components that operate on an

adjustment basis. However, a quick search of the index in the JDK 1.1.3 documentation package did not identify any other components that use the adjustment interface. Perhaps other adjustable objects are planned for future versions of the AWT. *(Note that as of Java 8, the **AdjustmentListener** interface is also implemented by the **AWTEventMulticaster** class. However, that class is not discussed in this lesson.)*

The sample program

This section presents a discussion of the program followed by the program listing.

Discussion

This program places a **Scrollbar** object and a **TextField** object in a **Frame** object as shown in Figure 1 .

Note: Figure 1. Graphic screen output for Event18.



Whenever the *bubble* in the **Scrollbar** is moved using any of the five available physical methods for moving the bubble, the *value* of the **Scrollbar** *(which normally should represent the position of the center of the bubble*) is extracted from the event object and displayed in the TextField object. *(Certain arithmetic corrections are required to cause the value to represent the center of the bubble because the actual value returned represents the left-hand edge of the bubble for a horizontal **Scrollbar** .)*

Also, whenever the bubble is moved, several other pieces of information are extracted from the object and displayed on the command-line screen. This includes the identification of the adjustable object and the type of adjustable event.

Note also that although five different initialization parameters are included as parameters to the **Scrollbar** constructor, other important initialization parameters such as **BlockIncrement** and **UnitIncrement** are not included in the parameter list. They must be set following instantiation using methods such as **setBlockIncrement** .

Experimentation indicates that the default value for **BlockIncrement** is ten units and the default value for **UnitIncrement** is one unit.

Reiterating what was mentioned earlier, the **Scrollbar** listener object is not added as a *ScrollbarListener* but rather is defined by implementing the **AdjustmentListener** interface and then added as an **AdjustmentListener** .

The **AdjustmentListener** interface declares only one method that may be overridden: **adjustmentValueChanged** .

This method receives an object of type **AdjustmentEvent** that *(as of Java 8)* provides five methods that can be used to obtain information about the event.

- The **getAdjustable** method of the **AdjustmentEvent** class returns information containing the identification of the object that generated the event.
- The **getAdjustmentType** method returns information containing the identification of the type of event (click the buttons on the Scrollbar, drag the bubble, etc.).
- The **paramString** method contains a variety of information which overlaps the two above methods to some extent.
- The **getValue** method returns the value produced by the adjustment.
- The **getValueIsAdjusting** method returns true if this is one of multiple adjustment events. *(This method was not defined in the class when this lesson was originally written in 1997.)*

Three of these five methods are used in the following program. The **paramString** and **getValueIsAdjusting** methods are not used.

There are many important comments in the program that add to this discussion, particularly with respect to bugs in the **Scrollbar** component, and arithmetic adjustments required to compensate for the width of the bubble. *(Note, however, that the comments related to bugs may no longer apply in Java 8 and beyond.)*

Interesting code fragments

I will discuss and explain this program in fragments. A complete listing is provided in Listing 9 .

Listing 1 shows the controlling class that contains the requisite **main** method. The **main** method instantiates an object of type **GUI** where all of the work is done.

Note: Listing 1. The controlling class for Event18.java.

```
import java.awt.*;
import java.awt.event.*;

class Event18{
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Event18
```

Listing 2 shows the beginning of the **GUI** class along with some instance variables used later in the program.

Note: Listing 2. Beginning of the GUI class.

```
class GUI{
  Scrollbar myScrollbar;
  TextField displayWindow;
  int bubbleWidth;//needs to be accessible by
event handler
```

Listing 3 shows the beginning of the constructor for the **GUI** class. The constructor begins by instantiating a **Frame** object to serve as a top-level window for the application.

Note: Listing 3. Beginning of the constructor for the GUI class.

```
  GUI(){
    Frame myFrame = new Frame("Copyright 1997,
R.G.Baldwin");

    bubbleWidth = 20;
    int initialPosition = 50 - bubbleWidth/2;
    int min = 0 - bubbleWidth/2;
    int max = 100 + bubbleWidth/2;
    myScrollbar = new
Scrollbar(Scrollbar.HORIZONTAL,
                      initialPosition, bubbleWidth,
min, max);
```

This is followed by instantiation of a horizontal **Scrollbar** object with a range from 0 to 100, an initial position at 50, a bubble width *(page size)* of 20, a unit increment of 2 *(see Listing 4)* , and a block increment of 15 *(see Listing 4)* .

The **Scrollbar** component does not center the bubble on the value. Rather, the left edge of the bubble is lined up with the value. The result is that values in the upper end of the range cannot be reached unless appropriate adjustments are made using half the bubble width. This distributes the unreachable values at each end of the Scrollbar. You can then set the *min* and *max* values to extend beyond the desired values by one-half the bubble width.

Listing 4 sets the unit increment to 2 and the block increment to 15. The unit increment is the amount of change resulting from clicking a button on the end of the **Scrollbar** . The block increment is the amount of change resulting from clicking between the bubble and the button on the end.

Note: Listing 4. Set the unit and block increments.

```
    myScrollbar.setBlockIncrement(15);
    myScrollbar.setUnitIncrement(2);
```

The code in Listing 5 should be very familiar to you by now. It really isn't very interesting at this point, and is included in this section solely for completeness.

Note: Listing 5. Complete the GUI.

```
    displayWindow = new TextField("Initial Text");
```

```
    displayWindow.setEditable(false);//make non-
editable

    //add components to the GUI
    myFrame.add("South", myScrollbar);
    myFrame.add("North", displayWindow);
    myFrame.setSize(300,75);
    myFrame.setVisible(true);

    //Instantiate a listener object for the
Scrollbar and
    // register it to receive notification of
adjustment
    // events.
    MyScrollbarListener myScrollbarListener =
      new MyScrollbarListener(this);
    //Note that the Scrollbar listener is not
added as a
    // ScrollbarListener but rather is added as an
    // AdjustmentListener.

myScrollbar.addAdjustmentListener(myScrollbarListe
ner);

    //Close Frame to terminate.
    myFrame.addWindowListener(new
MyWindowListener());
  }//end constructor
}//end class GUI
```

This fragment creates the display window using a **TextField** object. It also adds the two components to the **Frame** , sets the size of the **Frame** , and makes the whole thing visible.

The fragment also includes typical code to instantiate a listener object for the **Scrollbar** and register it to receive notification of adjustment events.

This is followed by typical code to register a window listener to terminate the program when the user clicks the close box on the **Frame** .

That ends the constructor and also ends the definition of the **GUI** class.

The code in Listing 6 shows the beginning of the **AdjustmentListener** class from which a listener object is instantiated and registered on the **Scrollbar** to listen for adjustment events. This fragment shows the constructor for the class that saves a reference to the **Scrollbar** .

Note: Listing 6. Beginning of class that implements AdjustmentListener.

```
class MyScrollbarListener implements
AdjustmentListener{
  GUI thisObject; //save ref to GUI object here

  MyScrollbarListener(GUI objectIn){//constructor
    thisObject = objectIn;
  }//end constructor
```

Listing 7 shows the beginning of the **adjustmentValueChanged** method, which is the only method declared by the **AdjustmentListener** interface. This method receives a parameter which is an object of type **AdjustmentEvent** and serves as the heart of this program.

Note: Listing 7. Beginning of the adjustmentValueChanged method.

```
  public void
adjustmentValueChanged(AdjustmentEvent e){
    int value;
    //Display the entire AdjustmentEvent object
    System.out.println(e);
    System.out.println("Adjustable = " +
e.getAdjustable());
    System.out.println("AdjustmentType = "
                                      +
e.getAdjustmentType());
```

This fragment uses the incoming object to obtain and display

- the incoming object,
- the source of the event, and
- the type of the event.

The text output shown in Figure 2 was produced by the above fragment when the button on the right end of the scrollbar was clicked with the program running under Java 8 and Windows 7.

Note: Figure 2. Output produced by clicking right scrollbar button.

```
java.awt.event.AdjustmentEvent[ADJUSTMENT_VALUE_CH
ANGED,adjType=UNIT_INCREMENT,v
alue=42,isAdjusting=false] on scrollbar0
Adjustable =
java.awt.Scrollbar[scrollbar0,4,51,292x20,val=42,v
is=20,min=-10,max
=110,horz,isAdjusting=false]
AdjustmentType = 1
```

```
value = 42
value = 42
```

Listing 8 uses the incoming object to get the *value* of the scrollbar and display it in the **TextField** object. The value should represent the position of the bubble *(it really represents the position of the left edge of the bubble, not the center).*

Note: Listing 8. Get and display the value of the scrollbar.

```
    value = e.getValue();//get and save the value

    //The following code is here to work around a
possible
    // bug in the Scrollbar object where small
negative
    // values are reported as very large positive
values
    // when the Scrollbar is adjusted by sliding
the bubble.
    // Note that this bug still exists in JDK
1.1.6 but has
    // been fixed in JDK 1.2.
    System.out.println("value = " + value);
    if(value > 65000) value = value - 65536;
    System.out.println("value = " + value);
    thisObject.displayWindow.setText("Value = "
                        + (value +
thisObject.bubbleWidth/2));

    //The following seems like a kludge but is
required to
```

```
    // make the scrollbar bubble stay put.
Otherwise, it
    // jumps back to the previous value when you
try to
    // move it.
    //Note that this statement is required for JDK
1.1.6
    // but is not required for the first release
of JDK 1.2
    thisObject.myScrollbar.setValue(value);
  }//end adjustmentValueChanged()

}//end class MyScrollbarListener
```

This fragment contains code to serve as a workaround for a bug in JDK 1.1.6 where small negative values are reported as very large positive values when the Scrollbar is adjusted by dragging the bubble. *(Note that this bug has been fixed in JDK 1.2.)*

The fragment also contains code to adjust the reported value to the center of the bubble instead of the left edge of the bubble by increasing the value by one-half the width of the bubble. The adjusted value is displayed in the **TextField** object.

It also contains code to resolve another bug which causes the bubble to jump back to its previous value when you attempt to drag it in JDK 1.1.6. *(This also has been fixed in JDK 1.2.)*

That concludes the interesting code fragments for this program. A complete listing is provided in the next section.

Complete program listing

Listing 9 provides a complete listing of the program. As mentioned earlier, some of the comments related to bugs may no longer apply in Java 8 and beyond.

Note: Listing 9. Complete listing of the program named Event18.java.

```
/*File Event18.java Copyright 1997, R.G.Baldwin

Revised 12/18/97
This program was designed to be compiled and
executed under
JDK 1.1.3 or later version.

In an earlier version of this program compiled
under
JDK 1.1, in February 1997, I reported numerous
bugs in the
JDK Scrollbar handling.  I have now recompiled the
program
and can report that most of the bugs seem to have
been fixed,
although a couple of bugs still seem to exist.

One remaining bug is that if you adjust the
scrollbar by
dragging the bubble into an area which would
report negative
values, large positive values are reported instead
of small
negative values (ie: something like 65536 instead
of -1).
This does not happen if the scrollbar is adjusted

by
clicking on the end buttons or in the area between
```

```
the
bubble and the buttons.  A numeric workaround was
implemented in this program to work around that
bug. Note
that the bug still exists in JDK 1.1.6 but was
fixed in the
first release of JDK 1.2.

Another possible bug is that the scrollbar seems
to be able
to report a value that is one unitIncrement too
large for
the specified maximum value and bubble width.
Note
that the bug still exists in JDK 1.1.6 but was
fixed in the
first release of JDK 1.2.

This program places a Scrollbar object and a
TextField
object in a Frame.  Whenever the bubble in the
Scrollbar is
moved using any of the five available methods for
moving
the bubble the value of the Scrollbar (which
should
represent the position of the bubble) is displayed
in the
TextField object.

Also, whenever the bubble is moved, several other
pieces of
information are displayed on the screen which
identify
various parameters of the adjustment.


Closing the frame terminates the program.
```

```
*/
//=============================================
=========
import java.awt.*;
import java.awt.event.*;

class Event18{
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Event18
//=============================================
==========
class GUI{
  Scrollbar myScrollbar;
  TextField displayWindow;
  int bubbleWidth;//needs to be accessible by
event handler

  GUI(){
    Frame myFrame = new Frame("Copyright 1997,
R.G.Baldwin");

    //Instantiate a horizontal Scrollbar object
with range
    // from 0 to 100, initial position at 50,
bubble width
    // (page size) of 20, unitIncrement of 2 and a
    // blockIncrement of 15.

    // The Scrollbar component does not center the
bubble on
    // the value.  Rather, the left edge of the
bubble is
    // lined up with the value.  The result is
that values
    // in the upper end of the range cannot be
```

```
reached
    // unless appropriate adjustments are made
using half
    // the bubble width.  This distributes the
unreachable
    // values at each end of the Scrollbar. You
can then
    // set the min and max values to extend beyond
the
    // desired values by one-half the bubble
width.
    bubbleWidth = 20;
    int initialPosition = 50 - bubbleWidth/2;
    int min = 0 - bubbleWidth/2;
    int max = 100 + bubbleWidth/2;
    myScrollbar = new
Scrollbar(Scrollbar.HORIZONTAL,
                      initialPosition, bubbleWidth,
min, max);

    //It would have been nice to have included
unitIncrement
    // and blockIncrement as parameters to the
constructor
    // along with the other constructor
parameteters. See
    // next two statements which set the
unitIncrement and
    // the blockIncrement.  The unitIncrement is
the amount
    // of change resulting from clicking a button
on the
    // end of the Scrollbar.  The blockIncrement
is the
    // amount of change resulting from clicking
between the
    // bubble and the button on the end.
```

```
    myScrollbar.setBlockIncrement(15);
    myScrollbar.setUnitIncrement(2);

    displayWindow = new TextField("Initial Text");
    displayWindow.setEditable(false);//make non-
editable

    //add components to the GUI
    myFrame.add("South", myScrollbar);
    myFrame.add("North", displayWindow);
    myFrame.setSize(300,75);
    myFrame.setVisible(true);

    //Instantiate a listener object for the
Scrollbar and
    // register it to receive notification of
adjustment
    // events.
    MyScrollbarListener myScrollbarListener =
      new MyScrollbarListener(this);
    //Note that the Scrollbar listener is not
added as a
    // ScrollbarListener but rather is added as an
    // AdjustmentListener.

myScrollbar.addAdjustmentListener(myScrollbarListe
ner);

    //Close Frame to terminate.
    myFrame.addWindowListener(new
MyWindowListener());
  }//end constructor
}//end class GUI
//===================================================
=========


//Note that unlike some other components which
```

```
have their
// own listener interface, this class does not
implement
// ScrollbarListener because there is no such
interface.
// Rather, the AdjustmentListener interface is
used as a
// listener interface for Scrollbars.
class MyScrollbarListener implements
AdjustmentListener{
  GUI thisObject; //save ref to GUI object here

  MyScrollbarListener(GUI objectIn){//constructor
    thisObject = objectIn;
  }//end constructor

  public void
adjustmentValueChanged(AdjustmentEvent e){
    int value;
    //Display the entire AdjustmentEvent object
    System.out.println(e);
    System.out.println("Adjustable = " +
e.getAdjustable());
    System.out.println("AdjustmentType = "
                                      +
e.getAdjustmentType());

    //Display the value of the Scrollbar object in
the
    // TextField.  The value should represent the
position
    // of the bubble.
    value = e.getValue();//get and save the value

    //The following code is here to work around a
possible
    // bug in the Scrollbar object where small
```

```
negative
    // values are reported as very large positive
values
    // when the Scrollbar is adjusted by sliding
the bubble.
    // Note that this bug still exists in JDK
1.1.6 but has
    // been fixed in JDK 1.2.
    System.out.println("value = " + value);
    if(value > 65000) value = value - 65536;
    System.out.println("value = " + value);
    thisObject.displayWindow.setText("Value = "
                        + (value +
thisObject.bubbleWidth/2));

    //The following seems like a kludge but is
required to
    // make the scrollbar bubble stay put.
Otherwise, it
    // jumps back to the previous value when you
try to
    // move it.
    //Note that this statement is required for JDK
1.1.6
    // but is not required for the first release
of JDK 1.2
    thisObject.myScrollbar.setValue(value);
  }//end adjustmentValueChanged()

}//end class MyScrollbarListener
//===================================================
=========
//Listener to terminate the program when the Frame
is
// closed.

class MyWindowListener extends WindowAdapter{
  public void windowClosing(WindowEvent e){
```

```
    System.exit(0);
  }//end windowClosing()
}//end class MyWindowListener
//===========================================
=========
```

Review

Q - All of the **Scrollbar** bugs were eliminated in JDK 1.1.3: True or False? If false, explain why.

A - False. JDK 1.1.3 appears to still have at least two Scrollbar bugs. One bug causes the *track* event *(which involves dragging the bubble in the Scrollbar object)* to return large positive values such as 65535 when it should be returning small negative values such as -1. The other apparent bug allows the different versions of the event to return a value larger than would be calculated using the minimum and maximum values along with the width of the bubble.

Q - The **Scrollbar** bugs listed above cause the **Scrollbar** component to be completely unusable: True or False? If false, explain why.

A - False. While these two bugs represent a significant inconvenience, it is probably possible to work around both of them by making numeric corrections.

Q - To create a listener object for a **Scrollbar** , you must implement the *ScrollbarListener* interface or extend a *ScrollbarAdapter* class: True or False? If false, explain why.

A - False. To create a listener object for a **Scrollbar** , you do not implement a *ScrollbarListener* interface or extend a *ScrollbarAdapter* class, because they do not exist. Instead, you implement an **AdjustmentListener** interface.

Q - To create a listener object for a **Scrollbar** , you can either implement the *AdjustmentListener* interface or extend the *AdjustmentAdapter* class: True or False? If False, explain why.

A - False. There is no *AdjustmentAdapter* class. It isn't needed because the **AdjustmentListener** interface declares only one method with the following signature:

Note:

```
public abstract void
adjustmentValueChanged(AdjustmentEvent e)
```

Q - The **adjustmentValueChanged** method is invoked when the value of the **Scrollbar** object is changed: True or False? If false, explain why.

A - True.

Q - There are five different types of adjustment events that correspond to the different methods of moving the bubble in a **Scrollbar** object: True or False. If false, explain why.

A - True.

Q - As with most of the other classes in the AWT, the *AdjustmentListener* interface declares a separate method for each type of adjustment event. Each of these methods must be implemented in any class that implements the *AdjustmentListener* interface: True or False? If false, explain why.

A - False. Only one method is declared in the *AdjustmentListener* interface. The name of the method is **adjustmentValueChanged** .

Q - The **adjustmentValueChanged** method receives an object of type **AdjustmentEvent** as a parameter, and the type of event is encoded into the object: True or False? If false, explain why.

A - True.

Q - In addition to the type of the event, additional information is also encoded into the object that is passed to the **adjustmentValueChanged** method. Give examples of two other kinds of information encoded into the object..

A - Information such as the *value* of the **Scrollbar** and the *name* of the **Scrollbar** component is also encoded into the object.

Q - The types of adjustment events are defined as static variables in the **AdjustmentEvent** class. What are the names of these variables, and how do they correspond to user actions on the **Scrollbar** object?

A - The names of the **static** variables and their relationship to the **Scrollbar** object are:

- UNIT_INCREMENT - click button on one end of the Scrollbar
- UNIT_DECREMENT - click button on the other end of the Scrollbar
- BLOCK_INCREMENT - click in space between bubble and one button
- BLOCK_DECREMENT - click in space between bubble and other button
- TRACK - drag the bubble

Q - The width of the bubble is also referred to by at least two other terms in the **Scrollbar** documentation. What are they?

A - The other terms used to describe the bubble width are *page size* and *visible* . These terms seem to be related to the use of a **Scrollbar** on the side or bottom of a window of text where the size of the bubble is indicative of the amount of text that is visible in the window in relation to the amount of text in the entire document..

Q - In JDK 1.1.3, the *value* of the **Scrollbar** object is represented by the center of the bubble: True or False? If false, explain why.

A - False. The *value* of the **Scrollbar** object is represented by an edge of the bubble. For example, the *value* is represented by the position of the left edge of the bubble for a horizontal **Scrollbar** . If you want the position of the center of the bubble to represent a *value* , you must perform a numeric correction *(involving half the width of the bubble)* on the *value* of the **Scrollbar** actually extracted from the event object.

Q - All necessary initialization parameters for a **Scrollbar** object are provided as parameters to one of the **Scrollbar** constructors: True or False: If false, explain why.

A - False. Although five different initialization parameters are included as parameters to the **Scrollbar** constructor, other important initialization parameters such as **BlockIncrement** and **UnitIncrement** are not included in the parameter list. They must be set following instantiation using methods such as **setBlockIncrement** .

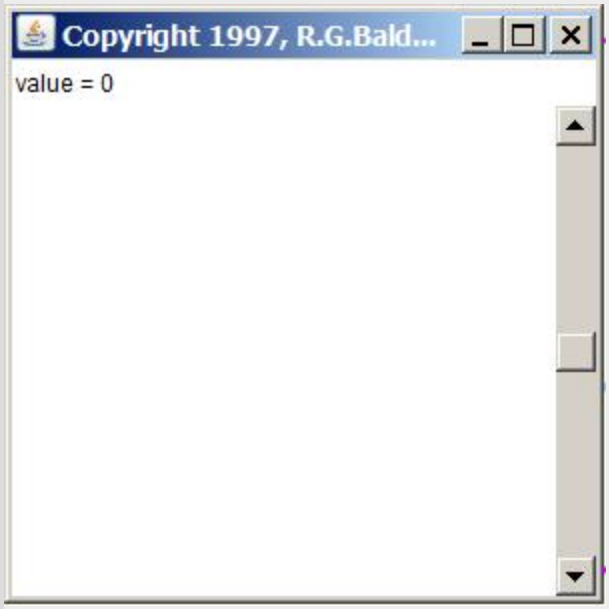
Q - The **AdjustmentEvent** class provides five methods that can be used to obtain information about the event. What are they and what kinds of information do they return?

A - The five methods and the information that they return are listed below:

- The **getAdjustable** method returns the identification of the object that generated the event.
- The **getAdjustmentType** method returns the identification of the type of event.
- The **paramString** method returns a variety of information which overlaps the two above methods to some extent.
- The **getValue** method returns the **value** produced by the adjustment.
- The **getValueIsAdjusting** method returns true if this is one of multiple adjustment events. *(This method was not defined in the class when this lesson was originally written in 1997.)*

Q - Write an application that meets the specifications given in the comments in the program shown in Listing 10 . The graphic screen output produced by the program is shown in Figure 3 .

Note: Figure 3. Graphic screen output from SampProg125.java.



Note: Listing 10. Complete listing of the program named SampProg125.java.

```
/*File SampProg125.java  from lesson 90
Copyright 1997, R. G. Baldwin
Without viewing the following solution, write a
Java
application that meets the specifications given
below.

To accommodate this program, you will need to be
using
```

```
JDK 1.1.3 or a later version.

Write an application that places a vertical
Scrollbar and a
Label on a Frame.  The Label is used to display
the
position of the center of the bubble in the
Scrollbar.

The Scrollbar values should range from
approximately -50
to +50 (exact end points are difficult to
achieve).

The width of the bubble should be 10 units.

When you click on the buttons at the end of the
Scrollbar,
the bubble should move by 3 units.

When you click in the area between the bubble and
the
buttons, the bubble should move by 6 units.

When the program starts, the bubble should be
centered
and the Label should indicate a bubble position of
0.

Closing the frame terminates the program.

End of specifications.
*/
//=============================================

=========
import java.awt.*;
import java.awt.event.*;
```

```
class SampProg125{
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class SampProg125
//===============================================
==========
class GUI{
  Scrollbar myScrollbar;
  Label displayWindow;
  int bubbleWidth;//needs to be accessible by
event handler

  GUI(){
    Frame myFrame = new Frame("Copyright 1997,
R.G.Baldwin");

    bubbleWidth = 10;
    int initialPosition = 0 - bubbleWidth/2;
    int min = -50 - bubbleWidth/2;
    int max = 50 + bubbleWidth/2;
    myScrollbar = new
Scrollbar(Scrollbar.VERTICAL,
                      initialPosition, bubbleWidth,
min, max);

    myScrollbar.setBlockIncrement(6);
    myScrollbar.setUnitIncrement(3);

    displayWindow = new Label("value = 0");

    //add components to the GUI
    myFrame.add("East", myScrollbar);

    myFrame.add("North", displayWindow);
    myFrame.setSize(300,300);
    myFrame.setVisible(true);
```

```
    //Instantiate a listener object for the
Scrollbar and
    // register it to receive notification of
adjustment
    // events.
    MyScrollbarListener myScrollbarListener =
                                new
MyScrollbarListener(this);
    //Note that the Scrollbar listener is not
added as a
    // ScrollbarListener but rather is added as an
    // AdjustmentListener.

myScrollbar.addAdjustmentListener(myScrollbarListe
ner);

    //Close Frame to terminate.
    myFrame.addWindowListener(new
MyWindowListener());
  }//end constructor
}//end class GUI
//=======================================
=========

//Note that unlike some other components which
have their
// own listener interface, this class does not
implement
// ScrollbarListener because there is no such
interface.
// Rather, the AdjustmentListener interface is
used as a
// listener interface for Scrollbars.

class MyScrollbarListener implements
AdjustmentListener{
  GUI thisObject; //save ref to GUI object here
```

```
  MyScrollbarListener(GUI objectIn){//constructor
    thisObject = objectIn;
  }//end constructor

  public void
adjustmentValueChanged(AdjustmentEvent e){
    int value;

    //Display the value of the Scrollbar object in
the
    // Label.  The value should represent the
position
    // of the bubble.
    value = e.getValue();//get and save the value

    //The following code is here to work around a
possible
    // bug in the Scrollbar object where small
negative
    // values are reported as very large positive
values
    // when the Scrollbar is adjusted by sliding
the bubble.
    if(value > 65000) value = value - 65536;
    thisObject.displayWindow.setText("Value = "
                        + (value +
thisObject.bubbleWidth/2));

    //The following seems like a kludge but is
required to
    // make the scrollbar bubble stay put.
Otherwise, it
    // jumps back to the previous value when you
try to
    // move it.
    thisObject.myScrollbar.setValue(value);
```

```
  }//end adjustmentValueChanged()

}//end class MyScrollbarListener
//===========================================
=========
//Listener to terminate the program when the Frame
is
// closed.
class MyWindowListener extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class MyWindowListener
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java090 Scrollbar (Adjustment) Event Handling
- File: Java090.htm
- Originally published: 1997

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java092 Mouse Motion Events
When we discussed mouse events in an earlier lesson, we didn't take into account that there are actually two different mouse interfaces: MouseListener and MouseMotionListener. In that lesson, we only discussed the implementation of methods in the MouseListener interface. In this lesson, we will also implement the mouseDragged() and the mouseMoved() methods in the MouseMotionListener interface to implement a program that can move a Label object on a Panel object by dragging the Label object with the mouse.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java092 Mouse Motion Events
- File: Java092.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java094 Using Abbreviated Inner Classes
JDK 1.1 was formally released on February 18, 1997. This lesson was originally written on February 27, 1997 using the software and documentation in JDK 1.1. One of the new features in JDK 1.1 in the capability to create inner classes and also to use a somewhat cryptic abbreviated syntax for the definition of anonymous classes and the instantiation of anonymous objects from those classes. The instantiation of Listener objects is a strong candidate for the use of anonymous classes and objects as provided by the inner class capability. An entire series of lessons will be dedicated to a discussion of inner classes. The purpose of this lesson is simply to introduce you to the concept so that you will know what you are seeing whenever you encounter code using the abbreviated inner class syntax.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If

not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java094 Using Abbreviated Inner Classes
- File: Java094.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for

those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java095 Container Events and More on Inner Classes
This lesson was originally written on December 10, 1997, using the JDK 1.1.3 download package. The purpose of this lesson is to illustrate the use of container events, and to further illustrate the use of inner classes, both named and anonymous.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java095 Container Events and More on Inner Classes
- File: Java095.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java096 Program-Generated Events and the AWTEventMulticaster Class
In this lesson, we take the subject of event handling one step further by teaching you how to create and dispatch events under program control which produce the same response as if the events were caused by actions of the user on visual components. In other words, you will learn how to create and dispatch events under program control which simulate the behavior of a user. For example, you might dispatch an event of a given type if a particular character were read from a file.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java096 Program-Generated Events and the AWTEventMulticaster Class
- File: Java096.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java097 Component Events
If you instantiate an object of type ComponentListener and register that object on an object that is a subclass of the Component class, methods of the listener object will be invoked whenever the object is hidden, moved, resized, or shown. Hiding and showing are accomplished by invoking the setVisible() method on the object with a parameter of false or true. Resizing and moving are accomplished by physically performing these actions on the visual representation of the object on the screen. Information regarding the event is passed into the ComponentListener methods in the form of an object of type ComponentEvent.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java097 Component Events
- File: Java097cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java099 Item Events
If you instantiate an object of type ItemListener and register that object on an object that has an addItemListener() method, the itemStateChanged() method of the listener object will be invoked whenever the state of the source object changes. An item event is a semantic event, and several different sources can multicast the event. I will explain the use of the ItemListener interface and ItemEvent class using radio buttons and checkboxes. Information regarding the event is passed into the itemStateChanged() method in the form of an object of type ItemEvent.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java099 Item Events
- File: Java099.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java100 Creating, Trapping, and Processing Custom Event Types
The purpose of this lesson is to illustrate one way that you can create your own custom event types. In other words, if you need a type of event that is not one of the standard types, it is possible for you to create a new type. Then you can cause events of that new type to be generated, trapped, and processed. This is an important aspect of Java Beans technology.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java100 Creating, Trapping, and Processing Custom Event Types
- File: Java100.cnx.htm
- Published: 01/14/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java101 Paint Events
The component-level paint event is a special type which is used to ensure that paint/update method calls are serialized along with the other events delivered from the event queue. This event is not designed to be used with the Event Listener model; programs should continue to override paint/update methods in order render themselves properly.

Table of contents

- Preface
- Tutorial link
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial link

Click here to download and view the PDF version of this page.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java101 Paint Events
- File: Java101.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java102 Handling Events in Extended Components without Listener Objects
The purpose of this lesson is to illustrate one way that you can create your own custom components by extending the existing components. When you extend an existing component, you need to handle the events associated with the new component.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java102 Handling Events in Extended Components without Listener Objects
- File: Java102.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java104 Posting Synthetic Events to the System Event Queue
The key ingredients for creating and posting event objects to the SystemEventQueue are explained in this lesson.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the

program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java104 Posting Synthetic Events to the System Event Queue
- File: Java104.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java105 Text Events
If you instantiate an object of type TextListener and register that object on an object that has an addTextListener() method, the textValueChanged() method of the listener object will be called whenever the text contents of the source object changes. A text event is a semantic event, and several different sources can multicast the event. I will explain the use of the TextListener interface and TextEvent class using a simple TextField object in this lesson.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java105 Text Events
- File: Java105.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java107 Understanding Action Objects in Java
The main purpose of this lesson is to help you to understand the use of Java Action objects, with particular emphasis on the changes introduced in Java version 1.3. Along the way, you will also learn something about bound properties in Java.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java107 Understanding Action Objects in Java
- File: Java107.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java110 The AWT Package, An Overview
This lesson is primarily a preview of what you can expect to find in several lessons immediately following this one. This and the next several lessons concentrate on the package java.awt where most of the functionality exists for providing graphics and the user interface to your application.

Note:
This page is included in the following books:

- Object-Oriented Programming (OOP) with Java
- Java Graphics

Table of contents

- Preface
- Tutorial link
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Java Graphics that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic,

you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial link

Click here to download and view the PDF version of this page.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java110 The AWT Package, An Overview
- File: Java110.cnx.htm
- Published: 01/12/16
- Revised: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java112 Placing Components in Containers, Absolute Coordinates
We can place components in containers using absolute position and size coordinates, or we can use any of several different layout managers to do the job. Using layout managers is considered by many to be the safer approach because this approach is designed to automatically compensate for differences in screen resolution and component size/shape between platforms. This lesson will discuss the placement of components into a container using absolute coordinates. Subsequent lessons will discuss the other possibilities mentioned above.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java112 Placing Components in Containers, Absolute Coordinates
- File: Java112.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java114 BorderLayout
In a previous lesson, we learned how to create a Frame object and place Button and Label components in the frame using absolute coordinates. We also learned that there can be some cross-platform compatibility problems with such an approach. This lesson looks at the first of several different approaches designed to place components in a container while minimizing cross-platform compatibility problems. This is accomplished using layout manager classes. In this lesson, we will concentrate on the BorderLayout class.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java114 BorderLayout
- File: Java114.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java116 FlowLayout
JDK 1.1 provides the FlowLayout manager, which solves cross-platform compatibility problems and allows you to place any (reasonable) number of components in a container. FlowLayout arranges components left to right until no more components fit on the same line. When no more components will fit on a line, a new line is started below the current line. Each line of components can either be centered, left aligned, or right aligned in the container. If the container is resized by the user at runtime, the locations of the components will be automatically adjusted to place the maximum possible number of components on the first and each successive line.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java116 FlowLayout
- File: Java116.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely

available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java118 GridLayout
In this lesson, we will learn how to use the GridLayout class for creating a layout manager. Perhaps more important, we will take another important step in learning about the User Interface. In particular, we will learn how to create composite user-interface objects by building up a combination of smaller objects.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java118 GridLayout
- File: Java118.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java120 CardLayout
In this lesson, we will learn how to use the CardLayout class for creating a layout manager. Also, we will continue learning how to create user-interface objects by building up a combination of smaller objects into the final composite object.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java120 CardLayout
- File: Java120.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1855 The KeyEventDispatcher in Java
Baldwin explains how to use KeyEventDispatcher objects to cause a KeyEvent fired by one component to be dispatched to a different component for processing.

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1855 The KeyEventDispatcher in Java
- File: Java1855cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java1856 The KeyEventPostProcessor in Java (Capturing Keyboard Strokes in Java)
The processing of a key event by the focus owner is not necessarily the end of the road for that event. Learn how to cause a chain of post processors to spring into action and to perform additional processing on the event before it finally dies

Table of contents

- Preface
- Tutorial and code links
- Miscellaneous

Preface

Over the years, I have published a large number of tutorials in the areas of computer programming and digital signal processing (DSP). As I have time available, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org .

In the meantime, this is one of the pages in a book titled Object-Oriented Programming (OOP) with Java that presents PDF versions of the original tutorials to make them readily available for Connexions users. When I have time available, I plan to update this tutorial and to re-publish it as a standard page at cnx.org .

This tutorial may contain internal links to other tutorials that I have written and published somewhere on the web. Those links may, or may not still be good. In any event, if you search cnx.org for the tutorial by title or by topic, you will probably find a clean copy of the referenced tutorial at cnx.org . If not, you can probably use a Google Advanced Search to find a copy somewhere on the web.

Tutorial and code links

Click here to download and view the PDF version of this page.

The representation of program code in PDF documents is often very unreliable. Click here to download a zip file containing a clean copy of the program code discussed in this tutorial.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java1856 The KeyEventPostProcessor in Java (Capturing Keyboard Strokes in Java)
- File: Java1856.cnx.htm
- Published: 01/15/16

Note: Disclaimers:
Financial : Although the Connexions website makes it possible for you to purchase a pre-printed version of the book containing this page, please be aware that the pre-printed version probably won't contain the contents of the PDF file referenced above .
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the pre-printed version of the book.
In the past, unknown individuals have copied my materials from cnx.org, converted them to Kindle books, and have placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of material that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java3102 Java OOP Modifying the World and SimpleTurtle Classes
Learn to make simple modifications to the World and SimpleTurtle classes that modify how a program that uses Ericson's library behaves.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listings

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the

figures and listings while you are reading about them.

Figures

- Figure 1 . The required graphic output image.
- Figure 2 . Required output on the command line screen.

Listings

- Listing 1 . Modification to load a jpg file by default.
- Listing 2 . Original array containing turtle colors.
- Listing 3 . Modified array of turtle colors.
- Listing 4 . Change the initial heading.
- Listing 5 . Source code for driver class named Prob01..
- Listing 6 . Source code for the modified World class..
- Listing 7 . Source code for the modified SimpleTurtle class..

Preview

Program specifications

Write a program named **Prob01** that uses the class definition shown in Listing 5 and Ericson's media library along with the image file named **Prob01.jpg** to produce the graphic output image shown in Figure 1 .

Figure 1 - The required graphic output image.



No new classes

You may **not** define any new classes to cause your program to behave as required, and you may not modify the class definition for the class named **Prob01** given in Listing 5 . You must copy and modify *(if necessary)* the following media classes to cause your program to produce the required output:

- World.java
- Turtle.java
- SimpleTurtle.java

Files in your folder

Your folder must contain only the following class files, source-code files, and image files:

- Prob01.class
- Prob01.java
- Prob01.jpg
- SimpleTurtle.class
- SimpleTurtle.java
- Turtle.class
- Turtle.java
- World.class

- World.java

Output text

In addition to the output image described above, your program must produce the output text on the command- line screen shown in Figure 2 .

Figure 2 - Required output on the command line screen.

```
Dick Baldwin
Picture, filename Prob01.jpg height 274 width
365
Dick Baldwin
Dick Baldwin
Dick Baldwin
Dick Baldwin
```

You must substitute your name for mine wherever my name appears in the image and on the command-line screen.

An analysis

As is often the case, the real challenge with this problem is to decide what needs to be done to satisfy the specifications.

Required modifications

By comparing the default behavior of the **World** and **SimpleTurtle** classes with the requirements of this program, it can be determined that the following modifications to the **World** and **SimpleTurtle** classes are

required to meet the specifications. *(Modification of the **Turtle** class is not required)* :

- Modify the **World** class to load a picture named **Prob01.jpg** as the default background for the world in place of the all-white **Picture** object.
- Modify the **World** class to display the student's name near the top of the image.
- Modify the **World** class to display the student's name and information about the picture on the command-line screen.
- Modify the **SimpleTurtle** class to change the initial heading for new turtle objects to northeast instead of north.
- Modify the **SimpleTurtle** class to change the order in which colors are assigned to new turtles as they are instantiated.

Discussion and sample code

Modifications to the World class

Ericson's **World** class was modified to cause it to load a jpg file by default instead of displaying a blank picture by default. It was also modified to cause it to display text on the background image and to display text on the command line screen. These changes are reflected in Figure 1 and Figure 2 .

A complete listing of the modified **World** class is shown in Listing 6 .

Modifying the code

The code used to accomplish the modifications described above is shown in Listing 1 .

Listing 1 - Modification to load a jpg file by default.

Listing 1 - Modification to load a jpg file by default.

```
//create the background picture

//picture = new Picture(width,height);

picture = new Picture("Prob01.jpg");
picture.addMessage("Dick Baldwin",10,20);
System.out.println(picture);
```

Note that one original statement was disabled and replaced by three new statements.

In addition, several other **println** statements were added at strategic locations within the **World** and **SimpleTurtle** classes *(not shown here)* to cause the student's name to appear multiple times in the text output shown in Figure 2 .

Meeting the requirements

These modifications to the **World** and **SimpleTurtle** classes met the following requirements established earlier under Analysis .

- Modify the **World** class to load a picture named **Prob01.jpg** as the default background for the world in place of the all-white **Picture** object.
- Modify the **World** class to display the student's name near the top of the image.
- Modify the **World** class to display the student's name and information about the picture on the command-line screen.

Modifications to the SimpleTurtle class

The **SimpleTurtle** class was modified to change the order in which colors are assigned to new turtle objects and to change the initial heading of the turtle from north to northeast.

A complete listing of the modified **SimpleTurtle** class is shown in Listing 7 near the end of the module.

Change the order of color assignment

Listing 2 declares and initializes an array of color data that is used in the original version of the **SimpleTurtle** class to assign colors to the turtles on a cyclical basis as they are instantiated.

Listing 2 - Original array containing turtle colors.

```
  /** array of colors to use for the turtles
*/
  private static Color[] colorArray =
    { Color.green,
      Color.cyan,
      new Color(204,0,204),
      Color.gray};
```

Listing 3 declares and initializes a modified version of the array of color data that is used to assign colors to the turtles as they are instantiated.

Listing 3 - Modified array of turtle colors.

```
  /** array of colors to use for the turtles
*/
  private static Color[] colorArray =
    { Color.cyan,
      new Color(204,0,204),
      Color.green,
      Color.gray};
```

Determining which color to use

The code that assigns colors to the turtles as they are instantiated keeps track of the number of turtle objects that have been instantiated.

An index is computed as the

turtle count modulus the length of the array .

The colors are extracted from the array on a cyclical basis as more and more turtle objects are instantiated.

Each time the number of turtles is evenly divisible by the length of the array, the index used to access colors from the array starts over at zero.

Meeting the requirements

This modification to the **SimpleTurtle** class accomplished the following requirement established earlier under Analysis .

- Modify the **SimpleTurtle** class to change the order in which colors are assigned to new turtles as they are instantiated.

Change the initial heading

Listing 4 modifies the initialization value for a variable named **heading** , which is used to establish the direction that the turtle is facing.

Listing 4 - Change the initial heading.

```
  /** heading angle */
  //THIS IS A MODIFICATION
  //private double heading = 0;//default faces
north
  private double heading = 45;// default faces
northeast
```

The default direction in the original version of the class is due north or zero degrees. The modified default direction is northeast or 45 degrees.

Meeting the requirements

This modification to the **SimpleTurtle** class accomplished the following requirement established earlier under Analysis .

- Modify the **SimpleTurtle** class to change the initial heading for new turtle objects to northeast instead of north.

That completes the required modifications that were established under Analysis .

Run the program

I encourage you to copy the code from Listing 5 , Listing 6 , and Listing 7 . Compile the code and execute it. Experiment with the code, making

changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click Prob01.jpg to download the required input image file.

Summary

You learned how to make simple modification to the **World** and **SimpleTurtle** classes that modify how a program that uses Ericson's library behaves.

What's next?

This module dealt with modifications to the **World** and **SimpleTurtle** Classes. The next module will deal with modifications to the **Turtle** and **SimpleTurtle** Classes.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Modifying the World and SimpleTurtle Classes
- File: Java3102.htm
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the classes discussed in this module are shown in Listing 5 , Listing 6 , and Listing 7 below.

Listing 5 - Source code for driver class named Prob01.

Listing 5 - Source code for driver class named Prob01.

```
public class Prob01{
  public static void main(String[] args){
    World mars = new World(200,250);
    Turtle joe = new Turtle(mars);
    joe.forward();
    Turtle bill = new Turtle(mars);
    bill.moveTo(50,125);
    Turtle sue = new Turtle(mars);
    sue.moveTo(150,125);
    Turtle tom = new Turtle(mars);
    tom.moveTo(100,225);
  }//end main method
}//end class Prob01
```

Listing 6 - Source code for the modified World class.

```
import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;


/*Note, this version of the World class was
modified to
 * cause it to load a jpg file by default instead
of
 * displaying a blank picture by default. 12/23/08
 */

/**
 * Class to represent a 2d world that can hold
```

```
turtles and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements
ModelDisplay
{
  ////////////////// fields
/////////////////////

  /** should automatically repaint when model
changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.WHITE;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;

  /** the list of turtles in the world */
  private List<Turtle> turtleList = new
ArrayList<Turtle>();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors
//////////////
```

```
  /**
   * Constructor that takes no arguments
   */
  public World()
  {
    // set up the world and make it visible
    initWorld(true);
  }

  /**
   * Constructor that takes a boolean to
   * say if this world should be visible
   * or not
   * @param visibleFlag if true will be visible
   * else if false will not be visible
   */
  public World(boolean visibleFlag)
  {
    initWorld(visibleFlag);
  }

  /**
   * Constructor that takes a width and height for
this
   * world
   * @param w the width for the world
   * @param h the height for the world
   */
  public World(int w, int h)
  {
    width = w;
    height = h;

    System.out.println("Dick Baldwin");
    // set up the world and make it visible
    initWorld(true);
```

```
  }

  ///////////////// methods
////////////////////////

  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new
Dimension(width,height));

    // create the background picture
    //THIS IS A MODIFICATION
    //picture = new Picture(width,height);
    picture = new Picture("Prob01.jpg");
    picture.addMessage("Dick Baldwin",10,20);
    System.out.println(picture);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }

  /**
   * Method to get the graphics context for
drawing on
   * @return the graphics context of the
```

```
background picture
   */
  public Graphics getGraphics() { return
picture.getGraphics(); }

  /**
   * Method to clear the background picture
   */
  public void clearBackground() { picture = new
Picture(width,height); }

  /**
   * Method to get the background picture
   * @return the background picture
   */
  public Picture getPicture() { return picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) { picture =
pict; }

  /**
   * Method to paint this component
   * @param g the graphics context
   */
  public synchronized void paintComponent(Graphics
g)
  {
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the background
```

```
image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
  }

  /**
   * Metod to get the last turtle in this world
   * @return the last turtle added to this world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle)
turtleList.get(turtleList.size() - 1);
  }


  /**
   * Method to add a model to this model displayer
   * @param model the model object to add
   */
  public void addModel(Object model)
  {
    turtleList.add((Turtle) model);
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to check if this world contains the
passed
   * turtle
   * @return true if there else false
   */
```

```
  public boolean containsTurtle(Turtle turtle)
  {
    return (turtleList.contains(turtle));
  }

  /**
   * Method to remove the passed object from the
world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }

  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify the
display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }
```

```
  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) {
autoRepaint = value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }

  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should
show or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the
world
   * @return a list of turtles in the world
```

```
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of
turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this
world
   * in the form of a string
   * @return a string of information about this
world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " +
getHeight() +
      " world with " + turtleList.size() + "
turtles in it.";
  }

} // end of World class
```

Listing 7 - Source code for the modified SimpleTurtle class.

```
import javax.swing.*;
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.Observer;
import java.util.Random;
```

```
 /*Note: This class was modified to change the
order of the
 * colors used for new turtle objects as well as
the
 * initial heading for the turtle. 12/23/08
 */

/**
 *Class that represents a Logo-style turtle.  The
turtle
 * starts off facing north.
 * A turtle can have a name, has a starting x and
y position,
 * has a heading, has a width, has a height, has a
visible
 * flag, has a body color, can have a shell color,
and has a pen.
 * The turtle will not go beyond the model display
or picture
 * boundaries.
 *
 * You can display this turtle in either a picture
or in
 * a class that implements ModelDisplay.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimpleTurtle
{
  ///////////////// fields
/////////////////////////

  /** count of the number of turtles created */
  private static int numTurtles = 0;
```

```
  /** array of colors to use for the turtles */
  //THIS IS A MODIFICATION
  //THE ORDER OF THE COLORS IN THE ARRAY HAS BEEN
MODIFIED
  private static Color[] colorArray = {
Color.cyan, new Color(204,0,204),Color.green,
Color.gray};

  /** who to notify about changes to this turtle
*/
  private ModelDisplay modelDisplay = null;

  /** picture to draw this turtle on */
  private Picture picture = null;

  /** width of turtle in pixels */
  private int width = 15;

  /** height of turtle in pixels */
  private int height = 18;

  /** current location in x (center) */
  private int xPos = 0;

  /** current location in y (center) */
  private int yPos = 0;

  /** heading angle */
  //THIS IS A MODIFICATION
  //private double heading = 0;// default is
facing north
  private double heading = 45;// default is facing
northeast

  /** pen to use for this turtle */
  private Pen pen = new Pen();
```

```
  /** color to draw the body in */
  private Color bodyColor = null;

  /** color to draw the shell in */
  private Color shellColor = null;

  /** color of information string */
  private Color infoColor = Color.black;

  /** flag to say if this turtle is visible */
  private boolean visible = true;

  /** flag to say if should show turtle info */
  private boolean showInfo = false;

  /** the name of this turtle */
  private String name = "No name";

  ////////////////// constructors
//////////////////

  /**
   * Constructor that takes the x and y position
for the
   * turtle
   * @param x the x pos
   * @param y the y pos
   */
  public SimpleTurtle(int x, int y)
  {
    xPos = x;
    yPos = y;
    bodyColor = colorArray[numTurtles %
colorArray.length];
    setPenColor(bodyColor);
    numTurtles++;
  }
```

```
  /**
   * Constructor that takes the x and y position
and the
   * model displayer
   * @param x the x pos
   * @param y the y pos
   * @param display the model display
   */
  public SimpleTurtle(int x, int y, ModelDisplay
display)
  {
    this(x,y); // invoke constructor that takes x
and y
    modelDisplay = display;
    display.addModel(this);
  }

  /**
   * Constructor that takes a model display and
adds
   * a turtle in the middle of it
   * @param display the model display
   */
  public SimpleTurtle(ModelDisplay display)
  {
    // invoke constructor that takes x and y
    this((int) (display.getWidth() / 2),
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);
    System.out.println("Dick Baldwin");
  }

  /**
   * Constructor that takes the x and y position
and the
```

```
   * picture to draw on
   * @param x the x pos
   * @param y the y pos
   * @param picture the picture to draw on
   */
  public SimpleTurtle(int x, int y, Picture
picture)
  {
    this(x,y); // invoke constructor that takes x
and y
    this.picture = picture;
    this.visible = false; // default is not to see
the turtle
  }

  /**
   * Constructor that takes the
   * picture to draw on and will appear in the
middle
   * @param picture the picture to draw on
   */
  public SimpleTurtle(Picture picture)
  {
    // invoke constructor that takes x and y
    this((int) (picture.getWidth() / 2),
         (int) (picture.getHeight() / 2));
    this.picture = picture;
    this.visible = false; // default is not to see
the turtle
  }

  ////////////////////// methods
/////////////////////////

  /**
   * Get the distance from the passed x and y
location
```

```
   * @param x the x location
   * @param y the y location
   */
  public double getDistance(int x, int y)
  {
    int xDiff = x - xPos;
    int yDiff = y - yPos;
    return (Math.sqrt((xDiff * xDiff) + (yDiff *
yDiff)));
  }

  /**
   * Method to turn to face another simple turtle
   */
  public void turnToFace(SimpleTurtle turtle)
  {
    turnToFace(turtle.xPos,turtle.yPos);
  }

   /**
   * Method to turn towards the given x and y
   * @param x the x to turn towards
   * @param y the y to turn towards
   */
  public void turnToFace(int x, int y)
  {
    double dx = x - this.xPos;
    double dy = y - this.yPos;
    double arcTan = 0.0;
    double angle = 0.0;

    // avoid a divide by 0
    if (dx == 0)
    {
      // if below the current turtle
      if (dy > 0)
        heading = 180;
```

```
      // if above the current turtle
      else if (dy < 0)
        heading = 0;
    }
    // dx isn't 0 so can divide by it
    else
    {
      arcTan = Math.toDegrees(Math.atan(dy / dx));
      if (dx < 0)
        heading = arcTan - 90;
      else
        heading = arcTan + 90;
    }

    // notify the display that we need to repaint
    updateDisplay();
  }

  /**
   * Method to get the picture for this simple
turtle
   * @return the picture for this turtle (may be
null)
   */
  public Picture getPicture() { return
this.picture; }

  /**
   * Method to set the picture for this simple
turtle
   * @param pict the picture to use
   */
  public void setPicture(Picture pict) {
this.picture = pict; }

  /**
```

```
   * Method to get the model display for this
simple turtle
   * @return the model display if there is one
else null
   */
  public ModelDisplay getModelDisplay() { return
this.modelDisplay; }

  /**
   * Method to set the model display for this
simple turtle
   * @param theModelDisplay the model display to
use
   */
  public void setModelDisplay(ModelDisplay
theModelDisplay)
  { this.modelDisplay = theModelDisplay; }

  /**
   * Method to get value of show info
   * @return true if should show info, else false
   */
  public boolean getShowInfo() { return
this.showInfo; }

  /**
   * Method to show the turtle information string
   * @param value the value to set showInfo to
   */
  public void setShowInfo(boolean value) {
this.showInfo = value; }

  /**
   * Method to get the shell color
   * @return the shell color
   */
  public Color getShellColor()
```

```
  {
    Color color = null;
    if (this.shellColor == null && this.bodyColor
!= null)
      color = bodyColor.darker();
    else color = this.shellColor;
    return color;
  }

  /**
   * Method to set the shell color
   * @param color the color to use
   */
  public void setShellColor(Color color) {
this.shellColor = color; }

  /**
   * Method to get the body color
   * @return the body color
   */
  public Color getBodyColor() { return
this.bodyColor; }

  /**
   * Method to set the body color which
   * will also set the pen color
   * @param color the color to use
   */
  public void setBodyColor(Color color)
  {
    this.bodyColor = color;
    setPenColor(this.bodyColor);
  }

  /**
   * Method to set the color of the turtle.
   * This will set the body color
```

```
   * @param color the color to use
   */
  public void setColor(Color color) {
this.setBodyColor(color); }

  /**
   * Method to get the information color
   * @return the color of the information string
   */
  public Color getInfoColor() { return
this.infoColor; }

  /**
   * Method to set the information color
   * @param color the new color to use
   */
  public void setInfoColor(Color color) {
this.infoColor = color; }

  /**
   * Method to return the width of this object
   * @return the width in pixels
   */
  public int getWidth() { return this.width; }

  /**
   * Method to return the height of this object
   * @return the height in pixels
   */
  public int getHeight() { return this.height; }

  /**
   * Method to set the width of this object
   * @param theWidth in width in pixels
   */
  public void setWidth(int theWidth) { this.width
= theWidth; }
```

```
  /**
   * Method to set the height of this object
   * @param theHeight the height in pixels
   */
  public void setHeight(int theHeight) {
this.height = theHeight; }

  /**
   * Method to get the current x position
   * @return the x position (in pixels)
   */
  public int getXPos() { return this.xPos; }

  /**
   * Method to get the current y position
   * @return the y position (in pixels)
   */
  public int getYPos() { return this.yPos; }

  /**
   * Method to get the pen
   * @return the pen
   */
  public Pen getPen() { return this.pen; }

  /**
   * Method to set the pen
   * @param thePen the new pen to use
   */
  public void setPen(Pen thePen) { this.pen =
thePen; }

  /**
   * Method to check if the pen is down
   * @return true if down else false
   */
```

```
  public boolean isPenDown() { return
this.pen.isPenDown(); }

  /**
   * Method to set the pen down boolean variable
   * @param value the value to set it to
   */
  public void setPenDown(boolean value) {
this.pen.setPenDown(value); }

  /**
   * Method to lift the pen up
   */
  public void penUp() {
this.pen.setPenDown(false);}

  /**
   * Method to set the pen down
   */
  public void penDown() {
this.pen.setPenDown(true);}

  /**
   * Method to get the pen color
   * @return the pen color
   */
  public Color getPenColor() { return
this.pen.getColor(); }

  /**
   * Method to set the pen color
   * @param color the color for the pen ink
   */
  public void setPenColor(Color color) {
this.pen.setColor(color); }

  /**
```

```
   * Method to set the pen width
   * @param width the width to use in pixels
   */
  public void setPenWidth(int width) {
this.pen.setWidth(width); }

  /**
   * Method to get the pen width
   * @return the width of the pen in pixels
   */
  public int getPenWidth() { return
this.pen.getWidth(); }

  /**
   * Method to clear the path (history of
   * where the turtle has been)
   */
  public void clearPath()
  {
    this.pen.clearPath();
  }

  /**
   * Method to get the current heading
   * @return the heading in degrees
   */
  public double getHeading() { return
this.heading; }

  /**
   * Method to set the heading
   * @param heading the new heading to use
   */
  public void setHeading(double heading)
  {
    this.heading = heading;
  }
```

```
  /**
   * Method to get the name of the turtle
   * @return the name of this turtle
   */
  public String getName() { return this.name; }

  /**
   * Method to set the name of the turtle
   * @param theName the new name to use
   */
  public void setName(String theName)
  {
    this.name = theName;
  }

  /**
   * Method to get the value of the visible flag
   * @return true if visible else false
   */
  public boolean isVisible() { return
this.visible;}

  /**
   * Method to hide the turtle (stop showing it)
   * This doesn't affect the pen status
   */
  public void hide() { this.setVisible(false); }

  /**
   * Method to show the turtle (doesn't affect
   * the pen status
   */
  public void show() { this.setVisible(true); }

  /**
   * Method to set the visible flag
```

```
   * @param value the value to set it to
   */
  public void setVisible(boolean value)
  {
    // if the turtle wasn't visible and now is
    if (visible == false && value == true)
    {
      // update the display
      this.updateDisplay();
    }

    // set the visibile flag to the passed value
    this.visible = value;
  }

  /**
   * Method to update the display of this turtle
and
   * also check that the turtle is in the bounds
   */
  public synchronized void updateDisplay()
  {
    // check that x and y are at least 0
    if (xPos < 0)
      xPos = 0;
    if (yPos < 0)
      yPos = 0;

    // if picture
    if (picture != null)
    {
      if (xPos >= picture.getWidth())
        xPos = picture.getWidth() - 1;
      if (yPos >= picture.getHeight())
        yPos = picture.getHeight() - 1;
      Graphics g = picture.getGraphics();
      paintComponent(g);
```

```
    }
    else if (modelDisplay != null)
    {
      if (xPos >= modelDisplay.getWidth())
        xPos = modelDisplay.getWidth() - 1;
      if (yPos >= modelDisplay.getHeight())
        yPos = modelDisplay.getHeight() - 1;
      modelDisplay.modelChanged();
    }
  }

  /**
   * Method to move the turtle foward 100 pixels
   */
  public void forward() { forward(100); }

  /**
   * Method to move the turtle forward the given
number of pixels
   * @param pixels the number of pixels to walk
forward in the heading direction
   */
  public void forward(int pixels)
  {
    int oldX = xPos;
    int oldY = yPos;

    // change the current position
    xPos = oldX + (int) (pixels *
Math.sin(Math.toRadians(heading)));
    yPos = oldY + (int) (pixels * -
Math.cos(Math.toRadians(heading)));

    // add a move from the old position to the new
position to the pen
    pen.addMove(oldX,oldY,xPos,yPos);
```

```
    // update the display to show the new line
    updateDisplay();
  }

  /**
   * Method to go backward by 100 pixels
   */
  public void backward()
  {
    backward(100);
  }

  /**
   * Method to go backward a given number of
pixels
   * @param pixels the number of pixels to walk
backward
   */
  public void backward(int pixels)
  {
    forward(-pixels);
  }

  /**
   * Method to move to turtle to the given x and y
location
   * @param x the x value to move to
   * @param y the y value to move to
   */
  public void moveTo(int x, int y)
  {
    this.pen.addMove(xPos,yPos,x,y);
    this.xPos = x;
    this.yPos = y;
    this.updateDisplay();
  }
```

```
  /**
   * Method to turn left
   */
  public void turnLeft()
  {
   this.turn(-90);
  }

  /**
   * Method to turn right
   */
  public void turnRight()
  {
    this.turn(90);
  }

  /**
   * Method to turn the turtle the passed degrees
   * use negative to turn left and pos to turn
right
   * @param degrees the amount to turn in degrees
   */
  public void turn(int degrees)
  {
    this.heading = (heading + degrees) % 360;
    this.updateDisplay();
  }

  /**
   * Method to draw a passed picture at the
current turtle
   * location and rotation in a picture or model
display
   * @param dropPicture the picture to drop
   */
  public synchronized void drop(Picture
dropPicture)
```

```
  {
    Graphics2D g2 = null;

    // only do this if drawing on a picture
    if (picture != null)
      g2 = (Graphics2D) picture.getGraphics();
    else if (modelDisplay != null)
      g2 = (Graphics2D)
modelDisplay.getGraphics();

    // if g2 isn't null
    if (g2 != null)
    {

      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate to turtle heading and translate to
xPos and yPos

g2.rotate(Math.toRadians(heading),xPos,yPos);

      // draw the passed picture

g2.drawImage(dropPicture.getImage(),xPos,yPos,null
);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);

      //  draw the pen
      pen.paintComponent(g2);
    }
  }

  /**
```

```
   * Method to paint the turtle
   * @param g the graphics context to paint on
   */
  public synchronized void paintComponent(Graphics
g)
  {
    // cast to 2d object
    Graphics2D g2 = (Graphics2D) g;

    // if the turtle is visible
    if (visible)
    {
      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate the turtle and translate to xPos
and yPos

g2.rotate(Math.toRadians(heading),xPos,yPos);

      // determine the half width and height of
the shell
      int halfWidth = (int) (width/2); // of shell
      int halfHeight = (int) (height/2); // of
shell
      int quarterWidth = (int) (width/4); // of
shell
      int thirdHeight = (int) (height/3); // of
shell
      int thirdWidth = (int) (width/3); // of
shell

      // draw the body parts (head)
      g2.setColor(bodyColor);
      g2.fillOval(xPos - quarterWidth,
                  yPos - halfHeight - (int)
```

```
(height/3),
                  halfWidth, thirdHeight);
      g2.fillOval(xPos - (2 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos - (int) (1.6 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (1.3 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (0.9 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);


      // draw the shell
      g2.setColor(getShellColor());
      g2.fillOval(xPos - halfWidth,
                  yPos - halfHeight, width,
height);

      // draw the info string if the flag is true
      if (showInfo)
        drawInfoString(g2);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);
    }

    //  draw the pen
    pen.paintComponent(g);
  }

  /**
   * Method to draw the information string
   * @param g the graphics context
```

```
   */
  public synchronized void drawInfoString(Graphics
g)
  {
    g.setColor(infoColor);
    g.drawString(this.toString(),xPos + (int)
(width/2),yPos);
  }

  /**
   * Method to return a string with informaiton
   * about this turtle
   * @return a string with information about this
object
   */
  public String toString()
  {
    return this.name + " turtle at " + this.xPos +
", " +
      this.yPos + " heading " + this.heading +
".";
  }

} // end of class
```

-end-

Java3104 Java OOP Modifications to the Turtle and SimpleTurtle Classes
Learn how to modify Ericson's Turtle and SimpleTurtle classes for a variety of purposes.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the

figures and listings while you are reading about them.

Figures

- Figure 1 . Required screen output.
- Figure 2 . Required text output.

Listings

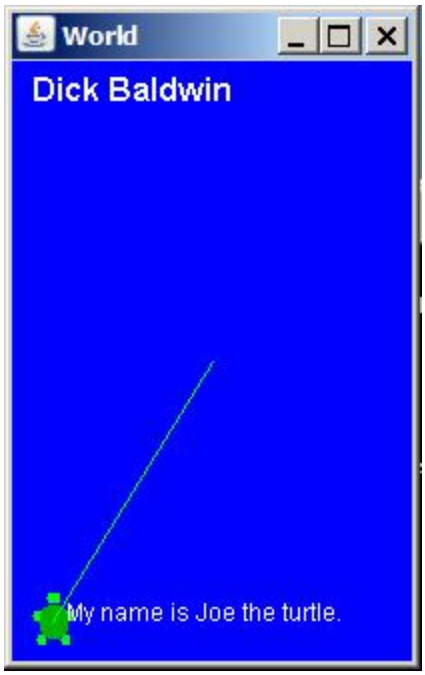
- Listing 1 . Modified Turtle constructor. .
- Listing 2 . Modified SimpleTurtle constructor.
- Listing 3 . Modified toString method.
- Listing 4 . Source code for the class named Prob02.
- Listing 5 . Modified Turtle class.
- Listing 6 . Modified SimpleTurtle class.

Preview

Program specifications

Write a program named **Prob02** that uses the class definition shown in Listing 4 and Ericson's media library to produce the graphic output image shown in Figure 1 .

Figure 1 - Required screen output.



No new classes allowed

You may not define any new classes to cause your program to behave as required, and you may not modify the class definition for the class named **Prob02** given in Listing 4 .

Files in your folder

You must copy and modify *(as necessary)* the media classes named **Turtle.java** and **SimpleTurtle.java** from Ericson's library to cause your program to produce the required output.

Your folder must contain only the following class files and source-code files:

- Prob02.class
- Prob02.java
- SimpleTurtle.class
- SimpleTurtle.java
- Turtle.class
- Turtle.java

Output text

In addition to the output image described above, your program must produce the text output shown in Figure 2 on the command- line screen

Figure 2 - Required text output.

```
Dick Baldwin
My name is Joe the turtle.
```

Required modifications

By comparing the default behavior of the **Turtle** and **SimpleTurtle** classes with the requirements of this program, it can be determined that the following modifications to the **Turtle** and **SimpleTurtle** classes are required to meet the specifications.

1. Modify the **Turtle** class to cause the student's name to be displayed on the command line.
2. Modify the **Turtle** and **SimpleTurtle** classes to accept and save a **String** parameter in addition to the **World** parameter when the Turtle object is constructed.
3. Modify the **SimpleTurtle** class to cause the default background of the world to be BLUE.
4. Modify the **SimpleTurtle** class to cause the student's name to be displayed near the top of the **World** image.
5. Modify the **toString** method in the **SimpleTurtle** class to cause it to return the value of the **String** parameter whenever the **toString** method is called. This causes the **drawInfoString** method to display the string in place of its normal behavior. It also causes the last statement in Listing 4 to display the turtle's name.

Discussion and sample code

Modifications to the Turtle class

Ericson's **Turtle** class was modified according to the first two items listed above under required modifications .

A complete listing of the modified **Turtle** class is provided in Listing 5 near the end of the module.

Modification to the Turtle constructor

The **Turtle** class has several overloaded constructors. One of the constructors was modified to accept a **String** parameter in addition to the **World** parameter and pass the new parameter along to the superclass constructor. The code is shown in Listing 1 .

Listing 1 - Modified Turtle constructor.

```
public Turtle (ModelDisplay modelDisplay,
               String turtleName){
  // let the parent constructor handle it
  super(modelDisplay,turtleName);
  System.out.println("Dick Baldwin");
}
```

A **println** statement was also added to the modified constructor to cause it to display the student's name on the command line screen when the **Turtle** object is constructed as shown in Figure 2.

Modifications to the SimpleTurtle class

A complete listing of the modified **SimpleTurtle** class is shown in Listing 6 near the end of the module.

The superclass of the Turtle class

The **SimpleTurtle** class is the superclass of the **Turtle** class. Therefore, the **SimpleTurtle** class must be modified to accept the **String** parameter passed to the superclass in Listing 1 . This was accomplished by modifying one of the constructors of the **SimpleTurtle** class as shown in Listing 2 .

Listing 2 - Modified SimpleTurtle constructor.

Listing 2 - Modified SimpleTurtle constructor.

```
  String turtleName = null;

  public SimpleTurtle(ModelDisplay display,
                      String turtleName){

    // call constructor that takes x and y
    this((int) (display.getWidth() / 2),
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);

    //THIS IS THE MODIFICATION
    this.turtleName = turtleName;
    Picture picture = ((World)
(display)).getPicture();
    picture.setAllPixelsToAColor(Color.BLUE);
    picture.addMessage("Dick Baldwin",10,20);
  }
```

The modification is shown in the last four statements in Listing 2 . This modification satisfies items 2, 3, and 4 listed earlier under required modifications .

Modified toString method

Listing 3 shows the modified **toString** method that satisfies item 5 listed above under required modifications .

Listing 3 - Modified toString method.

```
  public String toString(){
    //return this.name + " turtle at " +
this.xPos + ", " +
    //      this.yPos + " heading " +
this.heading + ".";
    return "My name is " + turtleName + " the
turtle.";
  }//end toString
```

The original code was preserved as comments in Listing 3 , and the new modified code is shown below those comments.

Run the program

I encourage you to copy the code from Listing 4 , Listing 5 , and Listing 6 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

You learned how to:

1. Modify the **Turtle** class to cause the student's name to be displayed on the command line.
2. Modify the **Turtle** and **SimpleTurtle** classes to accept and save a **String** parameter in addition to the **World** parameter when the **Turtle** object is constructed.
3. Modify the **SimpleTurtle** class to cause the default background of the world to be BLUE.
4. Modify the **SimpleTurtle** class to cause the student's name to be displayed near the top of the **World** image.

5. Modify the **toString** method in the **SimpleTurtle** class to cause it to return the value of the **String** parameter whenever the **toString** method is called. This causes the **drawInfoString** method to display the string in place of its normal behavior. It also causes the last statement in Listing 4 to display the turtle's name.

What's next?

In the next module, you will learn how to incorporate GUI components into a **World** object. In particular, you will learn how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Modifications to the Turtle and SimpleTurtle Classes
- File: Java3104.htm
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the programs discussed in this module are shown below.

Listing 4 - Source code for the class named Prob02.

Listing 4 - Source code for the class named Prob02.

```
import java.awt.Color;

public class Prob02{
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars,"Joe");
    joe.moveTo(20,280);
    joe.setInfoColor(Color.WHITE);
    joe.setShowInfo(true);
    System.out.println(joe);
  }//end main method
}//end class Prob02
```

Listing 5 - Modified Turtle class.

```
/*12/23/0812/23/08 This class and the class named
SimpleTurtle were modified to:

Accept and save a String parameter in addition to
the
World parameter when the Turtle object is
constructed.

Modify the toString method to cause it to return
the
value of the String parameter whenever the
toString
method is called. This causes the drawInfoString
method
to display the string in place of its normal
behavior.

Cause the default background of the world to be
```

```
BLUE.

Cause the student's name to be displayed near the
top of
the World image.

Cause the student's name as well as the turtle's
name to
be displayed on the command line.
*/

/**
 * Class that represents a turtle which is similar
to a Logo turtle.
 * This class inherts from SimpleTurtle and is for
students
 * to add methods to.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Turtle extends SimpleTurtle
{
  ////////////////// constructors
//////////////////////

  /** Constructor that takes the x and y and a
picture to
   * draw on
   * @param x the starting x position
   * @param y the starting y position
   * @param picture the picture to draw on
   */
  public Turtle (int x, int y, Picture picture)
  {
    // let the parent constructor handle it
    super(x,y,picture);
```

```
  }

  /** Constructor that takes the x and y and a
model
   * display to draw it on
   * @param x the starting x position
   * @param y the starting y position
   * @param modelDisplayer the thing that displays
the model
   */
  public Turtle (int x, int y,
                 ModelDisplay modelDisplayer)
  {
    // let the parent constructor handle it
    super(x,y,modelDisplayer);
  }

  //THIS IS A MODIFICATION
  //The following constructor was modified to
accept and
  // save a String parameter and pass it to the
superclass
  // constructor.
  /** Constructor that takes the model display
   * @param modelDisplay the thing that displays
the model
   */
  public Turtle (ModelDisplay modelDisplay,
                 String turtleName){
    // let the parent constructor handle it
    super(modelDisplay,turtleName);
    System.out.println("Dick Baldwin");
  }

  /**
   * Constructor that takes a picture to draw on
   * @param p the picture to draw on
```

```
   */
  public Turtle (Picture p)
  {
    // let the parent constructor handle it
    super(p);
  }

  ////////////////////// methods
//////////////////////


} // this } is the end of class Turtle, put all
new methods before this
```

Listing 6 - Modified SimpleTurtle class.

```
import javax.swing.*;
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.Observer;
import java.util.Random;

/*12/23/08 This class and the class named Turtle
were
modified to:
Accept and save a String parameter in addition to
the
World parameter when the Turtle object is
constructed.

Modify the toString method to cause it to return
the
value of the String parameter whenever the
toString
method is called. This causes the drawInfoString
method
```

```
to display the string in place of its normal
behavior.

Cause the default background of the world to be
BLUE.

Cause the student's name to be displayed near the
top of
the World image.

Cause the student's name as well as the turtle's
name to
be displayed on the command line.
 */

/**
 * Class that represents a Logo-style turtle.  The
turtle
 * starts off facing north.
 * A turtle can have a name, has a starting x and
y position,
 * has a heading, has a width, has a height, has a
visible
 * flag, has a body color, can have a shell color,
and has a pen.
 * The turtle will not go beyond the model display
or picture
 * boundaries.
 *
 * You can display this turtle in either a picture
or in
 * a class that implements ModelDisplay.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimpleTurtle
```

```
{
  ///////////////// fields
//////////////////////

  /** count of the number of turtles created */
  private static int numTurtles = 0;

  /** array of colors to use for the turtles */
  private static Color[] colorArray = {
Color.green, Color.cyan, new Color(204,0,204),
Color.gray};

  /** who to notify about changes to this turtle
*/
  private ModelDisplay modelDisplay = null;

  /** picture to draw this turtle on */
  private Picture picture = null;

  /** width of turtle in pixels */
  private int width = 15;

  /** height of turtle in pixels */
  private int height = 18;

  /** current location in x (center) */
  private int xPos = 0;

  /** current location in y (center) */
  private int yPos = 0;

  /** heading angle */
  private double heading = 0;  // default is
facing north

  /** pen to use for this turtle */
  private Pen pen = new Pen();
```

```
  /** color to draw the body in */
  private Color bodyColor = null;

  /** color to draw the shell in */
  private Color shellColor = null;

  /** color of information string */
  private Color infoColor = Color.black;

  /** flag to say if this turtle is visible */
  private boolean visible = true;

  /** flag to say if should show turtle info */
  private boolean showInfo = false;

  /** the name of this turtle */
  private String name = "No name";

  ////////////////// constructors
//////////////////

  /**
   * Constructor that takes the x and y position
for the
   * turtle
   * @param x the x pos
   * @param y the y pos
   */
  public SimpleTurtle(int x, int y)
  {
    xPos = x;
    yPos = y;
    bodyColor = colorArray[numTurtles %
colorArray.length];
    setPenColor(bodyColor);
    numTurtles++;
```

```
  }

  /**
   * Constructor that takes the x and y position
and the
   * model displayer
   * @param x the x pos
   * @param y the y pos
   * @param display the model display
   */
  public SimpleTurtle(int x, int y, ModelDisplay
display)
  {
    this(x,y); // call constructor that takes x
and y
    modelDisplay = display;
    display.addModel(this);
  }
  //THIS IS A MODIFICATION
  //The following constructor was modified to
accept and
  // save a String parameter.
  String turtleName = null;
  /**
   * Constructor that takes a model display and
adds
   * a turtle in the middle of it
   * @param display the model display
   */
  public SimpleTurtle(ModelDisplay display,
                      String turtleName){

    // call constructor that takes x and y
    this((int) (display.getWidth() / 2),
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);
```

```
    this.turtleName = turtleName;
    Picture picture = ((World)
(display)).getPicture();
    //THIS IS A MODIFICATION
    picture.setAllPixelsToAColor(Color.BLUE);
    picture.addMessage("Dick Baldwin",10,20);
  }

  /**
   * Constructor that takes the x and y position
and the
   * picture to draw on
   * @param x the x pos
   * @param y the y pos
   * @param picture the picture to draw on
   */
  public SimpleTurtle(int x, int y, Picture
picture)
  {
    this(x,y); // call constructor that takes x
and y
    this.picture = picture;
    this.visible = false; // default is not to see
the turtle
  }

  /**
   * Constructor that takes the
   * picture to draw on and will appear in the
middle
   * @param picture the picture to draw on
   */
  public SimpleTurtle(Picture picture)
  {
    // call constructor that takes x and y
    this((int) (picture.getWidth() / 2),
         (int) (picture.getHeight() / 2));
```

```
    this.picture = picture;
    this.visible = false; // default is not to see
the turtle
  }

  ////////////////////// methods
/////////////////////////

  /**
   * Get the distance from the passed x and y
location
   * @param x the x location
   * @param y the y location
   */
  public double getDistance(int x, int y)
  {
    int xDiff = x - xPos;
    int yDiff = y - yPos;
    return (Math.sqrt((xDiff * xDiff) + (yDiff *
yDiff)));
  }

  /**
   * Method to turn to face another simple turtle
   */
  public void turnToFace(SimpleTurtle turtle)
  {
    turnToFace(turtle.xPos,turtle.yPos);
  }

   /**
   * Method to turn towards the given x and y
   * @param x the x to turn towards
   * @param y the y to turn towards
   */
  public void turnToFace(int x, int y)
  {
```

```
    double dx = x - this.xPos;
    double dy = y - this.yPos;
    double arcTan = 0.0;
    double angle = 0.0;

    // avoid a divide by 0
    if (dx == 0)
    {
      // if below the current turtle
      if (dy > 0)
        heading = 180;

      // if above the current turtle
      else if (dy < 0)
        heading = 0;
    }
    // dx isn't 0 so can divide by it
    else
    {
      arcTan = Math.toDegrees(Math.atan(dy / dx));
      if (dx < 0)
        heading = arcTan - 90;
      else
        heading = arcTan + 90;
    }

    // notify the display that we need to repaint
    updateDisplay();
  }

  /**
   * Method to get the picture for this simple
turtle
   * @return the picture for this turtle (may be
null)
   */
  public Picture getPicture() { return
```

```
this.picture; }

  /**
   * Method to set the picture for this simple
turtle
   * @param pict the picture to use
   */
  public void setPicture(Picture pict) {
this.picture = pict; }

  /**
   * Method to get the model display for this
simple turtle
   * @return the model display if there is one
else null
   */
  public ModelDisplay getModelDisplay() { return
this.modelDisplay; }

  /**
   * Method to set the model display for this
simple turtle
   * @param theModelDisplay the model display to
use
   */
  public void setModelDisplay(ModelDisplay
theModelDisplay)
  { this.modelDisplay = theModelDisplay; }

  /**
   * Method to get value of show info
   * @return true if should show info, else false
   */
  public boolean getShowInfo() { return
this.showInfo; }

  /**
```

```
   * Method to show the turtle information string
   * @param value the value to set showInfo to
   */
  public void setShowInfo(boolean value) {
this.showInfo = value; }

  /**
   * Method to get the shell color
   * @return the shell color
   */
  public Color getShellColor()
  {
    Color color = null;
    if (this.shellColor == null && this.bodyColor
!= null)
      color = bodyColor.darker();
    else color = this.shellColor;
    return color;
  }

  /**
   * Method to set the shell color
   * @param color the color to use
   */
  public void setShellColor(Color color) {
this.shellColor = color; }

  /**
   * Method to get the body color
   * @return the body color
   */
  public Color getBodyColor() { return
this.bodyColor; }

  /**
   * Method to set the body color which
   * will also set the pen color
```

```
   * @param color the color to use
   */
  public void setBodyColor(Color color)
  {
    this.bodyColor = color;
    setPenColor(this.bodyColor);
  }

  /**
   * Method to set the color of the turtle.
   * This will set the body color
   * @param color the color to use
   */
  public void setColor(Color color) {
this.setBodyColor(color); }

  /**
   * Method to get the information color
   * @return the color of the information string
   */
  public Color getInfoColor() { return
this.infoColor; }

  /**
   * Method to set the information color
   * @param color the new color to use
   */
  public void setInfoColor(Color color) {
this.infoColor = color; }

  /**
   * Method to return the width of this object
   * @return the width in pixels
   */
  public int getWidth() { return this.width; }

  /**
```

```
   * Method to return the height of this object
   * @return the height in pixels
   */
  public int getHeight() { return this.height; }

  /**
   * Method to set the width of this object
   * @param theWidth in width in pixels
   */
  public void setWidth(int theWidth) { this.width
= theWidth; }

  /**
   * Method to set the height of this object
   * @param theHeight the height in pixels
   */
  public void setHeight(int theHeight) {
this.height = theHeight; }

  /**
   * Method to get the current x position
   * @return the x position (in pixels)
   */
  public int getXPos() { return this.xPos; }

  /**
   * Method to get the current y position
   * @return the y position (in pixels)
   */
  public int getYPos() { return this.yPos; }

  /**
   * Method to get the pen
   * @return the pen
   */
  public Pen getPen() { return this.pen; }
```

```
  /**
   * Method to set the pen
   * @param thePen the new pen to use
   */
  public void setPen(Pen thePen) { this.pen =
thePen; }

  /**
   * Method to check if the pen is down
   * @return true if down else false
   */
  public boolean isPenDown() { return
this.pen.isPenDown(); }

  /**
   * Method to set the pen down boolean variable
   * @param value the value to set it to
   */
  public void setPenDown(boolean value) {
this.pen.setPenDown(value); }

  /**
   * Method to lift the pen up
   */
  public void penUp() {
this.pen.setPenDown(false);}

  /**
   * Method to set the pen down
   */
  public void penDown() {
this.pen.setPenDown(true);}

  /**
   * Method to get the pen color
   * @return the pen color
   */
```

```
  public Color getPenColor() { return
this.pen.getColor(); }

  /**
   * Method to set the pen color
   * @param color the color for the pen ink
   */
  public void setPenColor(Color color) {
this.pen.setColor(color); }

  /**
   * Method to set the pen width
   * @param width the width to use in pixels
   */
  public void setPenWidth(int width) {
this.pen.setWidth(width); }

  /**
   * Method to get the pen width
   * @return the width of the pen in pixels
   */
  public int getPenWidth() { return
this.pen.getWidth(); }

  /**
   * Method to clear the path (history of
   * where the turtle has been)
   */
  public void clearPath()
  {
    this.pen.clearPath();
  }

  /**
   * Method to get the current heading
   * @return the heading in degrees
   */
```

```
  public double getHeading() { return
this.heading; }

  /**
   * Method to set the heading
   * @param heading the new heading to use
   */
  public void setHeading(double heading)
  {
    this.heading = heading;
  }

  /**
   * Method to get the name of the turtle
   * @return the name of this turtle
   */
  public String getName() { return this.name; }

  /**
   * Method to set the name of the turtle
   * @param theName the new name to use
   */
  public void setName(String theName)
  {
    this.name = theName;
  }

  /**
   * Method to get the value of the visible flag
   * @return true if visible else false
   */
  public boolean isVisible() { return
this.visible;}

  /**
   * Method to hide the turtle (stop showing it)
   * This doesn't affect the pen status
```

```
   */
  public void hide() { this.setVisible(false); }

  /**
   * Method to show the turtle (doesn't affect
   * the pen status
   */
  public void show() { this.setVisible(true); }

  /**
   * Method to set the visible flag
   * @param value the value to set it to
   */
  public void setVisible(boolean value)
  {
    // if the turtle wasn't visible and now is
    if (visible == false && value == true)
    {
      // update the display
      this.updateDisplay();
    }

    // set the visibile flag to the passed value
    this.visible = value;
  }

  /**
   * Method to update the display of this turtle
and
   * also check that the turtle is in the bounds
   */
  public synchronized void updateDisplay()
  {
    // check that x and y are at least 0
    if (xPos < 0)
      xPos = 0;
    if (yPos < 0)
```

```
      yPos = 0;

    // if picture
    if (picture != null)
    {
      if (xPos >= picture.getWidth())
        xPos = picture.getWidth() - 1;
      if (yPos >= picture.getHeight())
        yPos = picture.getHeight() - 1;
      Graphics g = picture.getGraphics();
      paintComponent(g);
    }
    else if (modelDisplay != null)
    {
      if (xPos >= modelDisplay.getWidth())
        xPos = modelDisplay.getWidth() - 1;
      if (yPos >= modelDisplay.getHeight())
        yPos = modelDisplay.getHeight() - 1;
      modelDisplay.modelChanged();
    }
  }

  /**
   * Method to move the turtle foward 100 pixels
   */
  public void forward() { forward(100); }

  /**
   * Method to move the turtle forward the given
number of pixels
   * @param pixels the number of pixels to walk
forward in the heading direction
   */
  public void forward(int pixels)
  {
    int oldX = xPos;
    int oldY = yPos;
```

```
    // change the current position
    xPos = oldX + (int) (pixels *
Math.sin(Math.toRadians(heading)));
    yPos = oldY + (int) (pixels * -
Math.cos(Math.toRadians(heading)));

    // add a move from the old position to the new
position to the pen
    pen.addMove(oldX,oldY,xPos,yPos);

    // update the display to show the new line
    updateDisplay();
  }

  /**
   * Method to go backward by 100 pixels
   */
  public void backward()
  {
    backward(100);
  }

  /**
   * Method to go backward a given number of
pixels
   * @param pixels the number of pixels to walk
backward
   */
  public void backward(int pixels)
  {
    forward(-pixels);
  }

  /**
   * Method to move to turtle to the given x and y
location
```

```
   * @param x the x value to move to
   * @param y the y value to move to
   */
  public void moveTo(int x, int y)
  {
    this.pen.addMove(xPos,yPos,x,y);
    this.xPos = x;
    this.yPos = y;
    this.updateDisplay();
  }

  /**
   * Method to turn left
   */
  public void turnLeft()
  {
   this.turn(-90);
  }

  /**
   * Method to turn right
   */
  public void turnRight()
  {
    this.turn(90);
  }

  /**
   * Method to turn the turtle the passed degrees
   * use negative to turn left and pos to turn
right
   * @param degrees the amount to turn in degrees
   */
  public void turn(int degrees)
  {
    this.heading = (heading + degrees) % 360;
    this.updateDisplay();
```

```
  }

  /**
   * Method to draw a passed picture at the
current turtle
   * location and rotation in a picture or model
display
   * @param dropPicture the picture to drop
   */
  public synchronized void drop(Picture
dropPicture)
  {
    Graphics2D g2 = null;

    // only do this if drawing on a picture
    if (picture != null)
      g2 = (Graphics2D) picture.getGraphics();
    else if (modelDisplay != null)
      g2 = (Graphics2D)
modelDisplay.getGraphics();

    // if g2 isn't null
    if (g2 != null)
    {

      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate to turtle heading and translate to
xPos and yPos

g2.rotate(Math.toRadians(heading),xPos,yPos);

      // draw the passed picture

g2.drawImage(dropPicture.getImage(),xPos,yPos,null
```

```
);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);

      //  draw the pen
      pen.paintComponent(g2);
    }
  }

  /**
   * Method to paint the turtle
   * @param g the graphics context to paint on
   */
  public synchronized void paintComponent(Graphics
g)
  {
    // cast to 2d object
    Graphics2D g2 = (Graphics2D) g;

    // if the turtle is visible
    if (visible)
    {
      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate the turtle and translate to xPos
and yPos

g2.rotate(Math.toRadians(heading),xPos,yPos);

      // determine the half width and height of
the shell
      int halfWidth = (int) (width/2); // of shell
      int halfHeight = (int) (height/2); // of
shell
```

```
      int quarterWidth = (int) (width/4); // of
shell
      int thirdHeight = (int) (height/3); // of
shell
      int thirdWidth = (int) (width/3); // of
shell

      // draw the body parts (head)
      g2.setColor(bodyColor);
      g2.fillOval(xPos - quarterWidth,
                  yPos - halfHeight - (int)
(height/3),
                  halfWidth, thirdHeight);
      g2.fillOval(xPos - (2 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos - (int) (1.6 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (1.3 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (0.9 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);


      // draw the shell
      g2.setColor(getShellColor());
      g2.fillOval(xPos - halfWidth,
                  yPos - halfHeight, width,
height);

      // draw the info string if the flag is true
      if (showInfo)
        drawInfoString(g2);
```

```
      // reset the tranformation matrix
      g2.setTransform(oldTransform);
    }

    //  draw the pen
    pen.paintComponent(g);
  }

  /**
   * Method to draw the information string
   * @param g the graphics context
   */
  public synchronized void drawInfoString(Graphics
g)
  {
    g.setColor(infoColor);
    g.drawString(this.toString(),xPos + (int)
(width/2),yPos);
  }
  //This toString method was modified.
  /**
   * Method to return a string with informaiton
   * about this turtle
   * @return a string with information about this
object
   */
  //THIS IS A MODIFICATION
  //MODIFIED toString METHOD
  public String toString()
  {
//    return this.name + " turtle at " + this.xPos
+ ", " +
//      this.yPos + " heading " + this.heading +
".";
    return "My name is " + turtleName + " the
turtle.";
  }
```

```
} // end of class
```

-end-

Java3106 Java OOP Incorporating GUI Components into a World Object
Learn how to incorporate GUI components into a World object. In particular, learn how to add a JButton object to a World object and register an action listener on the button to control the behavior of the program.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Initial screen output.
- Figure 2 . Screen output after clicking the button.
- Figure 3 . Required text output.

Listings

- Listing 1 . The new getFrame method.
- Listing 2 . Prob03 class definition.
- Listing 3 . Beginning of the Prob03Runner class.
- Listing 4 . Beginning of the run method.
- Listing 5 . Register an action listener on the button.
- Listing 6 . Set the picture background to blue.
- Listing 7 . Display the student's name on the picture.
- Listing 8 . Add a turtle to the world.
- Listing 9 . Source code for the program named Prob03.
- Listing 10 . The modified World class.

Preview

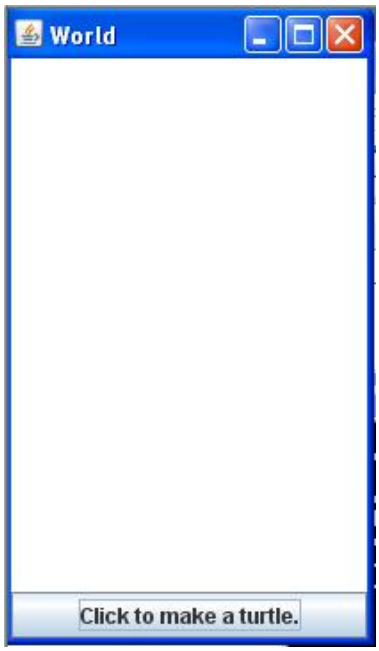
In this module, you will learn how to incorporate GUI components into a **World** object. In particular, you will learn how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

Program specifications

Write a program named **Prob03** that uses the **Prob03** class definition shown in Listing 2 and Ericson's media library to produce the graphic output images shown in Figure 1 and Figure 2 .

The image shown in Figure 1 is the image that appears on the screen when the program starts running.

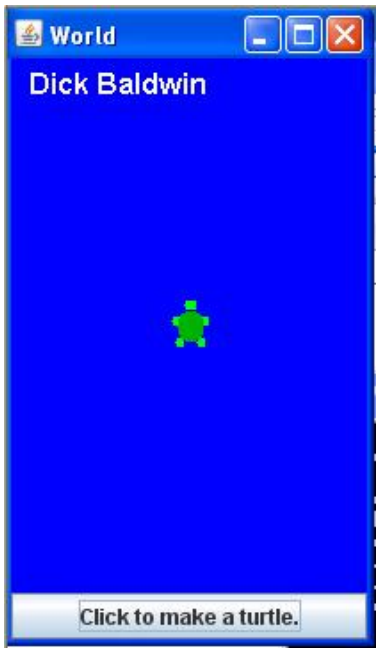
Figure 1 - Initial screen output.



Click the button

The image shown in Figure 2 is what you should see when you click the button at the bottom of the world.

Figure 2 - Screen output after clicking the button.



Modify Ericson's World class

You must copy and modify *(as necessary)* the media class named **World** to cause your program to produce the required output with the required behavior.

Add a JButton to the World

This program adds a **JButton** object to the SOUTH location of the **World** object as shown in Figure 1 and Figure 2 and registers an action listener on the button to control the behavior of the program.

Program behavior

The program initially displays an empty white world as shown in Figure 1 . When the user clicks the button, the world's background color changes to blue, a turtle appears in the center of the **World** , and the student's name appears near the top of the world.

Output text

In addition to the output images described above, your program must produce the text output shown in Figure 3 on the command- line screen

Figure 3 - Required text output.

```
Dick Baldwin
Dick Baldwin
Picture, filename None height 300 width 200
```

Analysis

A **World** object is actually a specialized use of a standard Java **JFrame** object. However, by default, the frame is not available to users of the **World** class. Therefore, in order to satisfy the requirements of this program, the **World** class must be modified to provide access to the frame.

Add a getFrame method

This program adds a method named **getFrame** to the **World** class. The **getFrame** method returns a reference to the **JFrame** object that is used to display the world. This makes it possible to treat **World** objects in much the same way that other **JFrame** objects are treated.

Add a button and pack the frame

The program uses the **JFrame** object's reference to add a **JButton** object to the SOUTH location of the **JFrame** . After adding the button, the program calls the **pack** method on the frame to cause the size of the frame to be automatically adjusted to accommodate both the **Picture** object that constitutes the background and the **JButton** object.

Discussion and sample code

Modification to the World class

A complete listing of the modified **World** class is provided in Listing 10 near the end of the module

Getting access to the frame

An object of the **World** class contains a private instance variable named **frame** that contains a reference to the **JFrame** object. Because it is private, however, it is not available to users of the **World** class. The *getter* method shown in Listing 1 was added to the **World** class to provide access to the **JFrame** .

Listing 1 - The new getFrame method.

```
public JFrame getFrame(){
  System.out.println("Dick Baldwin");
  return frame;
}//end getFrame
```

The new method also displays the student's name when the method is called, producing part of the text output in Figure 3 .

No other change to is required

This is the only change to Ericson's library that is required to write this program. Everything else in the program makes use of existing library classes with no modifications.

The Prob03Runner class

Will explain in fragments

I will explain this program in fragments. A complete listing is shown in Listing 9 near the end of the module.

The driver class

The driver class for this program is named **Prob03** . The definition of the driver class is shown in its entirety in Listing 2 .

Listing 2 - Prob03 class definition.

Listing 2 - Prob03 class definition.

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
```

The driver class simply instantiates a new object of a class named **Prob03Runner** and calls a method named **run** on that object.

Beginning of the Prob03Runner class

The beginning of the **Prob03Runner** class and the constructor for the class is shown in Listing 3 .

Listing 3 - Beginning of the Prob03Runner class.

```
class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Dick Baldwin");
  }//end constructor
```

As you can see, the constructor simply displays the student's name, providing some of the text output shown in Figure 3 .

Beginning of the run method

The beginning of the **run** method is shown in Listing 4 . This is where things start to get interesting.

Listing 4 - Beginning of the run method.

```
  public void run(){
    //This reference must be final because it is
    // referenced from within an anonymous class
    // definition.
    final World world = new World(200,300);

    //Get a reference to the JFrame object that is
used
    // to display the World.
    JFrame frame = world.getFrame();

    //Instantiate a new JButton object and add it to
the
    // SOUTH location in the JFrame object.
    JButton button = new JButton(
                                "Click to make a
turtle.");

frame.getContentPane().add(button,BorderLayout.SOUTH);


    frame.pack();//VERY IMPORTANT
```

A new World object

Listing 4 begins by instantiating a new object of the **World** class with a size of 200x300 pixels. The reference to the object is saved in a **final** variable named **world** .

A final reference variable

As the comment indicates, the variable must be **final** because it is referenced from within an anonymous class definition. I won't take the time to explain that here. I will simply refer you to my website where I have published several tutorial modules on anonymous classes.

The size of the world...

The purpose of specifying the size of the world when it is instantiated is to implicitly specify the size of the **Picture** object that forms the background for the world.

The size of the picture actually matches the specified dimensions. Therefore the actual size of the world is a little larger than the specified dimensions due to the borders that surround the picture.

Get a reference to the frame

After the **World** object is instantiated, the new **getFrame** method is called on the world's reference in order to get and save a reference to the frame.

A new JButton object

Then a new **JButton** object is instantiated. The reference to the **JButton** object is saved in the variable named **button** .

Add the button to the frame and pack it

Then the button is added to the SOUTH location in the frame and the **pack** method is called on the frame. Calling the **pack** method causes the size of the frame, *(and hence the size of the world)* to be adjusted so as to accommodate the picture in the CENTER of the frame and the button at the bottom *(SOUTH)* of the frame.

The final size of the world

After the button is added and the world is packed, the overall height of the world is quite a bit larger than the original dimensions. I measured it and found it to be about 209x361 pixels including borders.

The expansion in height is necessary to make room for the button. However, as you can see in Figure 3 , the size of the picture remains at 200x300 pixels.

Register an action listener on the button

I elected to use an *anonymous class* to register an action listener on the button. The purpose of the listener is to produce the desired behavior when the button is clicked.

Note, however, that there are other ways to register an action listener on the button and the student is not required to use an anonymous class for that purpose.

Beginning of the anonymous class

The definition of the anonymous listener class and the instantiation of the listener object begins in Listing 5 .

Listing 5 - Register an action listener on the button.

```
button.addActionListener(new ActionListener()
  {//Begin the class definition

    public void actionPerformed(ActionEvent e){
      Picture picture = world.getPicture();
      System.out.println(picture);
```

Unfamiliar with anonymous classes?

If you are unfamiliar with anonymous classes and action listeners, I will simply refer you to my website where I have published several tutorial modules on the topic. I have also published modules on the topic in this collection.

In a nutshell...

In a nutshell, however, the method named **actionPerformed** , which begins in Listing 5 , will be executed each time the user clicks the button in Figure 1 .

Behavior of the actionPerformed method

The code in Listing 5 gets and saves a reference to the **Picture** object that forms the background in the world object. Then it passes a copy of that reference to the **println** method, producing the third line of output text shown in Figure 3 .

Set the background picture to blue

Listing 6 calls the method named **setAllPixelsToAColor** on the **Picture** object passing the color BLUE as a parameter.

Listing 6 - Set the picture background to blue.

```
picture.setAllPixelsToAColor(Color.BLUE);
```

As you might expect, this causes the background of the world to turn from white to blue as shown in Figure 2 .

Display the student's name on the picture

Listing 7 calls the **addMessage** method on the picture to add the student's name near the upper-left corner of the world. *(See Figure 2 .)*

Listing 7 - Display the student's name on the picture.

```
picture.addMessage("Dick Baldwin",10,20);
```

Finally, Listing 8 instantiates a new **Turtle** object in the default color, with the default heading *(north)* , located in the default position, which is the center of the picture that constitutes the background image for the world.

Listing 8 - Add a turtle to the world.

```
          Turtle turtle = new Turtle(world);
        }//end actionPerformed

      }//end class definition
    );//end addActionListener

  }//end run
  //-------------------------------------------------
---//
}//end class Prob03Runner
```

Multiple clicks

If you click the button more than once, you will instantiate a new **Turtle** object and produce a line of output text on the command line screen with each click. The turtles will all be in the same location but they will cycle through four different color schemes.

Note that adding the turtle to the world causes the world to be repainted, eliminating the requirement to purposely repaint the world.

End the run method

Finally, Listing 8 signals the end of the **run** method, causing the **run** method to terminate and return control to the **main** method in Listing 2 . The **main** method terminates causing the program to terminate.

Run the program

I encourage you to copy the code from Listing 9 and Listing 10 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to incorporate GUI components into a **World** object. In particular, you learned how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

What's next?

In the next module, you will learn how to modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object. Then you will place a turtle object in a world and perform a series of maneuvers causing the turtle to draw a square spiral.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Incorporating GUI Components into a World Object
- File: Java3106.htm
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the programs discussed in this module are provided below.

Listing 9 - Source code for the program named Prob03.

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03


//=====================================================/
/
class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Dick Baldwin");
  }//end constructor
  //-----------------------------------------------
-//
  public void run(){
    //This reference must be final because it is
    // referenced from within an anonymous class
    // definition.
    final World world = new World(200,300);

    //Get a reference to the JFrame object that is used
    // to display the World.
    JFrame frame = world.getFrame();

    //Instantiate a new JButton object and add it to the
    // SOUTH location in the JFrame object.
    JButton button = new JButton(
                               "Click to make a
```

```
turtle.");

frame.getContentPane().add(button,BorderLayout.SOUTH);


    frame.pack();//VERY IMPORTANT


    //Use an anonymous class to register an action
    // listener on the button. Note that the student is
    // not required to use an anonymous class.
    button.addActionListener(new ActionListener()
      {//Begin the class definition
        public void actionPerformed(ActionEvent e){
          Picture picture = world.getPicture();
          System.out.println(picture);
          //Set picture background to blue.
          picture.setAllPixelsToAColor(Color.BLUE);
          //Display the student's name on the picture.
          picture.addMessage(
                         "Dick Baldwin",10,20);
          //Add a turtle to the world. This causes the
          // world to be repainted.
          Turtle turtle = new Turtle(world);
        }//end actionPerformed
      }//end class definition
    );//end addActionListener

  }//end run
  //-----------------------------------------------------
-//
}//end class Prob03Runner
```

Listing 10 - The modified World class.

```
import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;
```

```
/*12/23/08 Modified the World class. Added a method named
 *getFrame that returns a reference to the JFrame object
 *in which the turtles are displayed.
 */

/**
 * Class to represent a 2d world that can hold turtles
and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements
ModelDisplay
{
  ////////////////// fields /////////////////////

  /** should automatically repaint when model changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.white;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;

  /** the list of turtles in the world */
  private List<Turtle> turtleList = new ArrayList<Turtle>
();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors //////////////
```

```
/**
 * Constructor that takes no arguments
 */
public World()
{
  // set up the world and make it visible
  initWorld(true);
}

/**
 * Constructor that takes a boolean to
 * say if this world should be visible
 * or not
 * @param visibleFlag if true will be visible
 * else if false will not be visible
 */
public World(boolean visibleFlag)
{
  initWorld(visibleFlag);
}

/**
 * Constructor that takes a width and height for this
 * world
 * @param w the width for the world
 * @param h the height for the world
 */
public World(int w, int h)
{
  width = w;
  height = h;

  // set up the world and make it visible
  initWorld(true);
}

//////////////// methods ////////////////////////
/**
 *Method to return a reference to the JFrame.
 */
public JFrame getFrame(){
  System.out.println("Dick Baldwin");
```

```
    return frame;
  }//end getFrame

  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new Dimension(width,height));

    // create the background picture
    picture = new Picture(width,height);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }

  /**
   * Method to get the graphics context for drawing on
   * @return the graphics context of the background
picture
   */
  public Graphics getGraphics() { return
picture.getGraphics(); }

  /**
   * Method to clear the background picture
   */
  public void clearBackground() { picture = new
Picture(width,height); }

  /**
   * Method to get the background picture
```

```
   * @return the background picture
   */
  public Picture getPicture() { return picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) { picture = pict;
}

  /**
   * Method to paint this component
   * @param g the graphics context
   */
  public synchronized void paintComponent(Graphics g)
  {
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the background image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
  }

  /**
   * Metod to get the last turtle in this world
   * @return the last turtle added to this world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle) turtleList.get(turtleList.size() -
1);
  }
```

```
/**
 * Method to add a model to this model displayer
 * @param model the model object to add
 */
public void addModel(Object model)
{
  turtleList.add((Turtle) model);
  if (autoRepaint)
     repaint();
}

/**
 * Method to check if this world contains the passed
 * turtle
 * @return true if there else false
 */
public boolean containsTurtle(Turtle turtle)
{
  return (turtleList.contains(turtle));
}

/**
 * Method to remove the passed object from the world
 * @param model the model object to remove
 */
public void remove(Object model)
{
  turtleList.remove(model);
}

/**
 * Method to get the width in pixels
 * @return the width in pixels
 */
public int getWidth() { return width; }

/**
 * Method to get the height in pixels
 * @return the height in pixels
 */
public int getHeight() { return height; }
```

```
  /**
   * Method that allows the model to notify the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) { autoRepaint
= value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }

  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should show
or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }
```

```
  /**
   * Method to get the list of turtles in the world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this world
   * in the form of a string
   * @return a string of information about this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " + getHeight() +
      " world with " + turtleList.size() + " turtles in
it.";
  }

} // end of World class
```

-end-

Java3108 Java OOP Background Color, Text Color, Mouse Clicks, etc
Learn how to modify the SimplePicture class to control the color of the text that is placed on the image in a Picture object. Place a turtle object in a world and cause it to draw a square spiral.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listings

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Output image at startup.
- Figure 2 . Output image after ten mouse clicks.
- Figure 3 . Output image after eleven mouse clicks.

Listings

- Listing 1 . The new setMessageColor method.
- Listing 2 . The modified addMessage method.
- Listing 3 . Beginning of the Prob04Runner class.
- Listing 4 . Beginning of the anonymous listener class.
- Listing 5 . Add a turtle to the world.
- Listing 6 . Not the first click.
- Listing 7 . Process odd or even clicks.
- Listing 8 . Cause the program to terminate properly.
- Listing 9 . Source code for the program named Prob04.
- Listing 10 . Modified World class.
- Listing 11 . Modified SimplePicture class.

Preview

The program that I will explain in this module requires you to modify both the **World** class and the **SimplePicture** class from Ericson's media library.

Just as you did in an earlier module, you will modify the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object.

You will modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object.

Program specifications

Write a program named **Prob04** that uses the class definition for the **Prob04** class shown in Listing 9 along with Ericson's media library to produce the graphic output images shown in Figure 1 , Figure 2 , and Figure 3 .

Output image at startup

Figure 1 shows the output image when you first start the program.

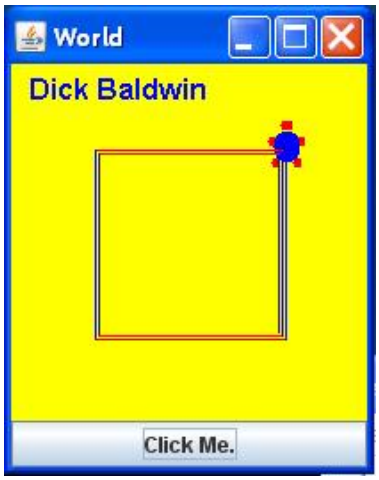
Figure 1 - Output image at startup.



Output image after ten mouse clicks

This program adds a **JButton** object to the SOUTH location of the **World** object as shown in Figure 1 . Figure 2 shows the output image after you click button ten times.

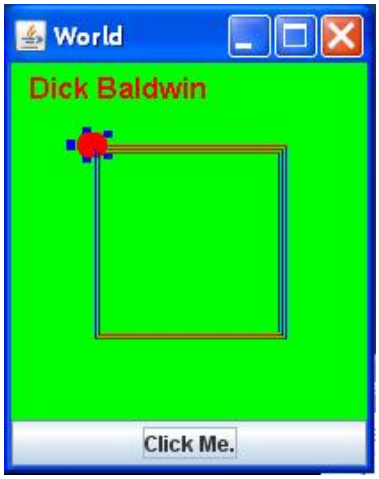
Figure 2 - Output image after ten mouse clicks.



Output image after eleven mouse clicks

Figure 3 shows the output image after you click button eleven times.

Figure 3 - Output image after eleven mouse clicks.



Operational description

The program initially displays an empty white world with a button at the bottom as shown in Figure 1 .

When the user clicks the button:

- The world's background color changes to green.
- A turtle appears near the bottom right corner of the World.
- The student's name appears near the top left corner of the world in red.
- The turtle has a blue body and a red shell.

Click the button again

When you click the button the second time:

- The background color changes to yellow.
- The student's name changes to blue.
- The turtle changes to a red body with a blue shell.
- The turtle turns 90 degrees left and moves forward 100 pixels plus the value of a click counter.
- The turtle leaves a blue trail.

Click the button another time

On the next click:

- The colors revert to the color scheme with the yellow background.
- The turtle turns 90 degrees left and moves forward 100 pixels plus the value of the click counter leaving a red trail.

Click the button repeatedly

This cycle repeats on each click with the turtle's trail drawing a square spiral of increasing size with red lines on the top and bottom of the spiral and blue lines on the right and left of the spiral.

Output text

In addition to the output images described above, your program must produce some output text on the command- line screen

Discussion and sample code

The driver class

The driver class named **Prob04** is shown at the beginning of Listing 9 near the end of the module. The **main** method simply instantiates a new object of the class named **Prob04Runner** , which I will explain later. The event driven behavior of the program is controlled by a listener object that is registered on the button in the constructor of the **Prob04Runner** class.

Modification of the World class

This program adds a method named **getFrame** to the **World** class. The method returns a reference to the **JFrame** object that is used to display the world.

The program uses that reference to add a **JButton** object to the SOUTH location of the **World** . I explained a modification very similar to this in an earlier module, so I won't repeat that explanation here.

A complete listing of the modified **World** class is provided in Listing 10 near the end of the module.

Modification of the SimplePicture class

This program modifies the **addMessage** method of the **SimplePicture** class to cause it to use a color variable named **messageColor** to set the color of the text. The modification also declares and initializes the private instance variable named **messageColor** .

The **SimplePicture** class was also modified to include a **setMessageColor** method that can be used to set the color value stored in the variable named **messageColor** .

The new setMessageColor method

The new method named **setMessageColor** that was added to the **SimplePicture** class is shown in Listing 1 .

Listing 1 - The new setMessageColor method.

```
public void setMessageColor(Color color){
  System.out.println("Dick Baldwin");
  messageColor = color;
}//end setMessageColor
```

Save the color value

The **setMessageColor** method saves the incoming color parameter in a private instance variable named **messageColor** that was added to the class.

The default value of the variable is **Color.WHITE** , thereby preserving the default behavior of the **addMessage** method.

You can view the new variable named **messageColor** in Listing 11 near the end of the module.

Display the student's name

The new **setMessageColor** method also causes my name to be displayed each time the method is called. This is of no operational value, but is useful during the testing stage of the modified class. This is part of the code that produces text output on the command line screen.

The modified addMessage method

The modified version of the **addMessage** method is shown in Listing 2 .

Listing 2 - The modified addMessage method.

```
 public void addMessage(
                      String message, int xPos, int
yPos){
   // get a graphics context to use to draw on the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // set the color to white
   //graphics2d.setPaint(Color.white);

   //modified by Baldwin on 12/23/08
   graphics2d.setPaint(messageColor);

   // set the font to Helvetica bold style and size
16
   graphics2d.setFont(new
Font("Helvetica",Font.BOLD,16));

   // draw the message
   graphics2d.drawString(message,xPos,yPos);

 }//end addMessage
```

The original statement shown in Listing 2 was disabled and replaced by the statement shown following the modification comment. This causes the text to be displayed on the image using the color stored in the new private instance variable named **messageColor** .

The Prob04Runner class

I will explain this program in fragments. A complete listing is shown in Listing 9 near the end of the module.

The driver class

The driver class for this program is named **Prob04** . As I mentioned earlier, you can view the class definition in its entirety near the beginning of Listing 9 .

Beginning of the Prob04Runner class

The **Prob04Runner** class begins in Listing 3 .

Listing 3 - Beginning of the Prob04Runner class.

```
class Prob04Runner{
  Turtle turtle = null;
  Picture picture = null;
  int counter = 0;
  World world = new World(200,200);
  JButton button = new JButton("Click Me.");

  public Prob04Runner(){
    System.out.println("Dick Baldwin");
    System.out.println(world.getPicture());

    //Get a reference to the JFrame object that is
used
    // to display the World.
    JFrame frame = world.getFrame();

    //Add the JButton object to the
    // SOUTH location in the JFrame object.

frame.getContentPane().add(button,BorderLayout.SOUTH);

    frame.pack();
```

Very familiar code

You should already be familiar with all of the code in Listing 3 . When the code in Listing 3 has finished executing, the image shown in Figure 1 should have appeared on the screen.

Waiting for an event

At this point, the program is essentially idle waiting for the user to either click the button at the bottom of Figure 1 , or click the red X-button in the upper-right corner of Figure 1 . *(More on the red X-button later.)*

Beginning of the anonymous listener class

This program uses an anonymous inner class to register an action listener on the button. The anonymous class begins in Listing 4 .

Listing 4 - Beginning of the anonymous listener class.

```
    button.addActionListener(new ActionListener()
      {//Begin the anonymous class definition

        public void actionPerformed(ActionEvent e){
          picture = world.getPicture();

          //Set picture background to green.
          picture.setAllPixelsToAColor(Color.GREEN);
          picture.setMessageColor(Color.RED);


          //Display the student's name on the
picture.
          picture.addMessage("Dick Baldwin",10,20);
```

The event handler method named actionPerformed

Once the listener object is instantiated from the anonymous class and registered on the button, the method named **actionPerformed** , which begins in Listing 4 , will be executed each time the button is clicked.

Get a reference to the background picture

The **actionPerformed** method begins by getting and saving a reference to the **Picture** object that provides the background image for the **World** object. By default, all of the pixels in this image are white, as shown in Figure 1 .

Set the background color to green

Then the method calls Ericson's standard method named **setAllPixelsToAColor** method to set the color of all of the background pixels to green.

Display student's name in red

Following that, the method calls the new **setMessageColor** method to set the text color to red, and calls the modified **addMessage** method to display my name in red near the upper-left corner of the image. Figure 3 shows an example of a green background and red text.

Add a turtle to the world

Listing 5 tests to determine if the variable named **turtle** that was declared in Listing 3 still contains null. If so, that means that this is the first time that the button has been clicked and the **Turtle** object has not yet been added to the world.

Listing 5 - Add a turtle to the world.

```
if(turtle == null){
  turtle = new Turtle(150,150,world);
  turtle.setHeading(90);
  turtle.setShellColor(Color.RED);
  turtle.setBodyColor(Color.BLUE);
```

Add a Turtle object and set its colors

Listing 5 instantiates a new **Turtle** object and adds it to the world near the lower-right corner facing due east *(90 degrees)* .

The shell color is set to red and the body color *(feet and head)* is set to blue. An example of the turtle with this color scheme is shown in Figure 3 .

Not the first click

If the conditional clause in Listing 5 returns false, that means that this is not the first time the button has been clicked and the else clause, which begins in Listing 6 will be executed.

Listing 6 - Not the first click.

```
}else{
  turtle.turnLeft();
  turtle.forward(100+counter);
```

Rotate and move

The **else** clause begins by causing the turtle to rotate to the left by a default angle of 90 degrees. Then the turtle moves forward by a distance equal to 100 plus the value of a counter that is incremented by one each time the button is clicked.

For example, on the second click of the button, the turtle moves toward the north drawing a blue line along the way. The default width of the line is one pixel and the default color of the line is the same as the shell color.

Process odd or even clicks

The behavior of the **actionPerformed** method at this point depends on whether the incremented value of the **counter** variable is even or odd. The code to accomplish this is shown in Listing 7 .

Listing 7 - Process odd or even clicks.

```
            if(counter++ %2 != 0){

picture.setAllPixelsToAColor(Color.GREEN);
              picture.setMessageColor(Color.RED);

              picture.addMessage(
                         "Dick Baldwin",10,20);
              turtle.setShellColor(Color.RED);
              turtle.setBodyColor(Color.BLUE);
            }else{

picture.setAllPixelsToAColor(Color.YELLOW);
              picture.setMessageColor(Color.BLUE);


              picture.addMessage(
                         "Dick Baldwin",10,20);
              turtle.setShellColor(Color.BLUE);
              turtle.setBodyColor(Color.RED);
            }//end else

            picture.addMessage(
                         "Dick Baldwin",10,20);
          }//end else

        }//end actionPerformed
      }//end class definition
    );//end addActionListener

  }//end constructor
  //-----------------------------------------------
---//

}//end class Prob04Runner
```

If the counter value is odd...

If the value of the counter (*before it is incremented -- note the post-increment operator)* is odd, the color scheme for the background, the message, and the turtle is set to that shown in Figure 3 with the green background.

If the counter value is even...

Otherwise, if the counter value is even, the color scheme is set to that shown in Figure 2 with the yellow background.

The end of several sections of code

Listing 7 also completes the **else** clause that began in Listing 6 .

In addition, Listing 7 signals the end of the **actionPerformed** method, the end of the anonymous class definition, the end of the constructor, and the end of the class named **Prob04Runner** .

Waiting for an event

As mentioned earlier, once the constructor finishes execution, the program becomes idle waiting for the user to either click the button at the bottom of Figure 1 , or click the red X-button in the upper-right corner of Figure 1 .

Does not terminate as expected

Normally a user would expect the program to terminate and return control to the operating system when the user clicks the red X-button in the upper-right corner of the last remaining window. However, this program does not do that. Instead, clicking this button simply hides the window and control is not returned to the operating system.

A programming oversight

This was a programming oversight on my part, which can be corrected by adding the second statement in Listing 8 to the definition of the **Prob04Runner** class immediately following the first statement shown in Listing 8 .

Listing 8 - Cause the program to terminate properly.

Listing 8 - Cause the program to terminate properly.

```
    JFrame frame = world.getFrame();

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

To understand why this is necessary to cause the program to terminate, I recommend that you visit the standard Sun javadocs and examine the description of the method named **setDefaultCloseOperation** in the **JFrame** class.

Run the program

I encourage you to copy the code from Listing 9 , Listing 10 , and Listing 11 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

Just as you did in an earlier module, you modified the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object.

You learned how to modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object.

Then you placed a turtle object in a world and performed a series of maneuvers causing the turtle to draw a square spiral.

What's next?

In the next module, you will learn how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Background Color, Text Color, Mouse Clicks, etc.
- File: Java3108.htm
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown below.

Listing 9 - Source code for the program named Prob04.

```
/*File Prob04 Copyright 2008 R.G.Baldwin
*******************************************************
/
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import java.awt.Toolkit;
```

```
public class Prob04{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob04Runner();
  }//end main method
}//end class Prob04
//===================================================/
/

class Prob04Runner{
  Turtle turtle = null;
  Picture picture = null;
  int counter = 0;
  World world = new World(200,200);
  JButton button = new JButton("Click Me.");

  public Prob04Runner(){
    System.out.println("Dick Baldwin");
    System.out.println(world.getPicture());

    //Get a reference to the JFrame object that is used
    // to display the World.
    JFrame frame = world.getFrame();

    //Add the JButton object to the
    // SOUTH location in the JFrame object.

frame.getContentPane().add(button,BorderLayout.SOUTH);

    frame.pack();

    //Use an anonymous class to register an action
    // listener on the button. Note that the student is
    // not required to use an anonymous class.
    button.addActionListener(new ActionListener()
      {//Begin the class definition
        public void actionPerformed(ActionEvent e){
          picture = world.getPicture();

          //Set picture background to green.
          picture.setAllPixelsToAColor(Color.GREEN);
```

```
        picture.setMessageColor(Color.RED);

        //Display the student's name on the picture.
        picture.addMessage(
                      "Dick Baldwin",10,20);
        //Add a turtle to the world. This causes the
        // world to be repainted.
        if(turtle == null){
          turtle = new Turtle(150,150,world);
          turtle.setHeading(90);
          turtle.setShellColor(Color.RED);
          turtle.setBodyColor(Color.BLUE);
        }else{
          turtle.turnLeft();
          turtle.forward(100+counter);
          if(counter++ %2 != 0){
            picture.setAllPixelsToAColor(Color.GREEN);
            picture.setMessageColor(Color.RED);

            picture.addMessage(
                      "Dick Baldwin",10,20);
            turtle.setShellColor(Color.RED);
            turtle.setBodyColor(Color.BLUE);
          }else{
            picture.setAllPixelsToAColor(Color.YELLOW);
            picture.setMessageColor(Color.BLUE);


            picture.addMessage(
                      "Dick Baldwin",10,20);
            turtle.setShellColor(Color.BLUE);
            turtle.setBodyColor(Color.RED);
          }//end else
          picture.addMessage(
                      "Dick Baldwin",10,20);
        }//end else
      }//end actionPerformed
    }//end class definition
  );//end addActionListener

}//end constructor
//-----------------------------------------------
```

```
-//

}//end class Prob04Runner
```

Listing 10 - Modified World class.

```
import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;

/*12/23/08 Modified the World class. Added a method named
 *getFrame that returns a reference to the JFrame object
 *in which the turtles are displayed.
 */

/**
 * Class to represent a 2d world that can hold turtles
and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements
ModelDisplay
{
  ////////////////// fields /////////////////////

  /** should automatically repaint when model changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.white;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;
```

```
  /** the list of turtles in the world */
  private List<Turtle> turtleList = new ArrayList<Turtle>
();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors ///////////////

  /**
   * Constructor that takes no arguments
   */
  public World()
  {
    // set up the world and make it visible
    initWorld(true);
  }

  /**
   * Constructor that takes a boolean to
   * say if this world should be visible
   * or not
   * @param visibleFlag if true will be visible
   * else if false will not be visible
   */
  public World(boolean visibleFlag)
  {
    initWorld(visibleFlag);
  }

  /**
   * Constructor that takes a width and height for this
   * world
   * @param w the width for the world
   * @param h the height for the world
   */
  public World(int w, int h)
  {
```

```
    width = w;
    height = h;

    // set up the world and make it visible
    initWorld(true);
  }

  ///////////////// methods ////////////////////////
  /**
   *Method to return a reference to the JFrame.
   */
  public JFrame getFrame(){
    System.out.println("Dick Baldwin");
    return frame;
  }//end getFrame

  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new Dimension(width,height));

    // create the background picture
    picture = new Picture(width,height);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }

  /**
   * Method to get the graphics context for drawing on
   * @return the graphics context of the background
```

```
picture
   */
  public Graphics getGraphics() { return
picture.getGraphics(); }

  /**
   * Method to clear the background picture
   */
  public void clearBackground() { picture = new
Picture(width,height); }

  /**
   * Method to get the background picture
   * @return the background picture
   */
  public Picture getPicture() { return picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) { picture = pict;
}

  /**
   * Method to paint this component
   * @param g the graphics context
   */
  public synchronized void paintComponent(Graphics g)
  {
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the background image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
```

```
  }

  /**
   * Metod to get the last turtle in this world
   * @return the last turtle added to this world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle) turtleList.get(turtleList.size() -
1);
  }


  /**
   * Method to add a model to this model displayer
   * @param model the model object to add
   */
  public void addModel(Object model)
  {
    turtleList.add((Turtle) model);
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to check if this world contains the passed
   * turtle
   * @return true if there else false
   */
  public boolean containsTurtle(Turtle turtle)
  {
    return (turtleList.contains(turtle));
  }

  /**
   * Method to remove the passed object from the world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }
```

```
  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) { autoRepaint
= value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
```

```
//    frame.setVisible(true);
//  }

  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should show
or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this world
   * in the form of a string
   * @return a string of information about this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " + getHeight() +
      " world with " + turtleList.size() + " turtles in
it.";
  }

} // end of World class
```

Listing 11 - Modified SimplePicture class.

```
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.swing.ImageIcon;
import java.awt.*;
import java.io.*;
import java.awt.geom.*;

/*
12/23/08 Modified the addMessage method to cause it to
use a color variable to set the color of the message.
Also provided a setMessageColor method to set the color
and a variable named messageColor to contain the color.
/*



/**
 * A class that represents a simple picture.  A simple
picture may have
 * an associated file name and a title.  A simple picture
has pixels,
 * width, and height.  A simple picture uses a
BufferedImage to
 * hold the pixels.  You can show a simple picture in a
 * PictureFrame (a JFrame).
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimplePicture implements DigitalPicture
{

  ////////////////////// Fields
//////////////////////
  /**
   * the color of the message
   */
   private Color messageColor = Color.WHITE;


  /**
   * the file name associated with the simple picture
   */
```

```
  private String fileName;

  /**
   * the title of the simple picture
   */
  private String title;

  /**
   * buffered image to hold pixels for the simple picture
   */
  private BufferedImage bufferedImage;

  /**
   * frame used to display the simple picture
   */
  private PictureFrame pictureFrame;

  /**
   * extension for this file (jpg or bmp)
   */
  private String extension;


 ////////////////////// Constructors
//////////////////////////

 /**
  * A Constructor that takes no arguments.  All fields
will be null.
  * A no-argument constructor must be given in order for
a class to
  * be able to be subclassed.  By default all subclasses
will implicitly
  * call this in their parent's no argument constructor
unless a
  * different call to super() is explicitly made as the
first line
  * of code in a constructor.
  */
 public SimplePicture()
 {this(200,100);}
```

```
 /**
  * A Constructor that takes a file name and uses the
file to create
  * a picture
  * @param fileName the file name to use in creating the
picture
  */
 public SimplePicture(String fileName)
 {

   // load the picture into the buffered image
   load(fileName);

 }

 /**
  * A constructor that takes the width and height desired
for a picture and
  * creates a buffered image of that size.  This
constructor doesn't
  * show the picture.
  * @param width the desired width
  * @param height the desired height
  */
 public  SimplePicture(int width, int height)
 {
   bufferedImage = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }

 /**
  * A constructor that takes the width and height desired
for a picture and
  * creates a buffered image of that size.  It also takes
the
  * color to use for the background of the picture.
  * @param width the desired width
  * @param height the desired height
```

```
  * @param theColor the background color for the picture
  */
 public  SimplePicture(int width, int height, Color
theColor)
 {
   this(width,height);
   setAllPixelsToAColor(theColor);
 }

 /**
  * A Constructor that takes a picture to copy
information from
  * @param copyPicture the picture to copy from
  */
 public SimplePicture(SimplePicture copyPicture)
 {
   if (copyPicture.fileName != null)
   {
      this.fileName = new String(copyPicture.fileName);
      this.extension = copyPicture.extension;
   }
   if (copyPicture.title != null)
      this.title = new String(copyPicture.title);
   if (copyPicture.bufferedImage != null)
   {
     this.bufferedImage = new
BufferedImage(copyPicture.getWidth(),

copyPicture.getHeight(), BufferedImage.TYPE_INT_RGB);
     this.copyPicture(copyPicture);
   }
 }

 /**
  * A constructor that takes a buffered image
  * @param image the buffered image
  */
 public SimplePicture(BufferedImage image)
 {
   this.bufferedImage = image;
   title = "None";
   fileName = "None";
```

```
    extension = "jpg";
  }

  ////////////////////////// Methods
 ////////////////////////////////
  /**
   * Method to set the color used for a message.
   */
   public void setMessageColor(Color color){
     System.out.println("Dick Baldwin");
     messageColor = color;
   }//end setMessageColor


  /**
   * Method to get the extension for this picture
   * @return the extendsion (jpg or bmp)
   */
  public String getExtension() { return extension; }


  /**
   * Method that will copy all of the passed source
picture into
   * the current picture object
   * @param sourcePicture  the picture object to copy
   */
  public void copyPicture(SimplePicture sourcePicture)
  {
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0, targetX = 0;
         sourceX < sourcePicture.getWidth() &&
         targetX < this.getWidth();
         sourceX++, targetX++)
    {
      // loop through the rows
      for (int sourceY = 0, targetY = 0;
           sourceY < sourcePicture.getHeight() &&
           targetY < this.getHeight();
```

```
          sourceY++, targetY++)
     {
       sourcePixel =
sourcePicture.getPixel(sourceX,sourceY);
       targetPixel = this.getPixel(targetX,targetY);
       targetPixel.setColor(sourcePixel.getColor());
     }
   }

 }

 /**
  * Method to set the color in the picture to the passed
color
  * @param color the color to set to
  */
 public void setAllPixelsToAColor(Color color)
 {
   // loop through all x
   for (int x = 0; x < this.getWidth(); x++)
   {
     // loop through all y
     for (int y = 0; y < this.getHeight(); y++)
     {
       getPixel(x,y).setColor(color);
     }
   }
 }

 /**
  * Method to get the buffered image
  * @return the buffered image
  */
 public BufferedImage getBufferedImage()
 {
    return bufferedImage;
 }

 /**
  * Method to get a graphics object for this picture to
use to draw on
  * @return a graphics object to use for drawing
```

```
   */
  public Graphics getGraphics()
  {
    return bufferedImage.getGraphics();
  }

  /**
   * Method to get a Graphics2D object for this picture
which can
   * be used to do 2D drawing on the picture
   */
  public Graphics2D createGraphics()
  {
    return bufferedImage.createGraphics();
  }

  /**
   * Method to get the file name associated with the
picture
   * @return  the file name associated with the picture
   */
  public String getFileName() { return fileName; }

  /**
   * Method to set the file name
   * @param name the full pathname of the file
   */
  public void setFileName(String name)
  {
    fileName = name;
  }

  /**
   * Method to get the title of the picture
   * @return the title of the picture
   */
  public String getTitle()
  { return title; }

  /**
   * Method to set the title for the picture
   * @param title the title to use for the picture
```

```
  */
 public void setTitle(String title)
 {
   this.title = title;
   if (pictureFrame != null)
       pictureFrame.setTitle(title);
 }

 /**
  * Method to get the width of the picture in pixels
  * @return the width of the picture in pixels
  */
 public int getWidth() { return bufferedImage.getWidth();
}

 /**
  * Method to get the height of the picture in pixels
  * @return  the height of the picture in pixels
  */
 public int getHeight() { return
bufferedImage.getHeight(); }

 /**
  * Method to get the picture frame for the picture
  * @return the picture frame associated with this
picture
  * (it may be null)
  */
 public PictureFrame getPictureFrame() { return
pictureFrame; }

 /**
  * Method to set the picture frame for this picture
  * @param pictureFrame the picture frame to use
  */
 public void setPictureFrame(PictureFrame pictureFrame)
 {
   // set this picture objects' picture frame to the
passed one
   this.pictureFrame = pictureFrame;
 }
```

```
 /**
  * Method to get an image from the picture
  * @return  the buffered image since it is an image
  */
 public Image getImage()
 {
   return bufferedImage;
 }

 /**
  * Method to return the pixel value as an int for the
given x and y location
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @return the pixel value as an integer (alpha, red,
green, blue)
  */
 public int getBasicPixel(int x, int y)
 {
    return bufferedImage.getRGB(x,y);
 }

 /**
  * Method to set the value of a pixel in the picture
from an int
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @param rgb the new rgb value of the pixel (alpha,
red, green, blue)
  */
 public void setBasicPixel(int x, int y, int rgb)
 {
   bufferedImage.setRGB(x,y,rgb);
 }

 /**
  * Method to get a pixel object for the given x and y
location
  * @param x  the x location of the pixel in the picture
  * @param y  the y location of the pixel in the picture
  * @return a Pixel object for this location
  */
```

```
 public Pixel getPixel(int x, int y)
 {
   // create the pixel object for this picture and the
given x and y location
   Pixel pixel = new Pixel(this,x,y);
   return pixel;
 }

 /**
  * Method to get a one-dimensional array of Pixels for
this simple picture
  * @return a one-dimensional array of Pixel objects
starting with y=0
  * to y=height-1 and x=0 to x=width-1.
  */
 public Pixel[] getPixels()
 {
   int width = getWidth();
   int height = getHeight();
   Pixel[] pixelArray = new Pixel[width * height];

   // loop through height rows from top to bottom
   for (int row = 0; row < height; row++)
     for (int col = 0; col < width; col++)
       pixelArray[row * width + col] = new
Pixel(this,col,row);

   return pixelArray;
 }



 /**
  * Method to load the buffered image with the passed
image
  * @param image  the image to use
  */
 public void load(Image image)
 {
   // get a graphics context to use to draw on the
buffered image
   Graphics2D graphics2d =
```

```
bufferedImage.createGraphics();

   // draw the image on the buffered image starting at
0,0
   graphics2d.drawImage(image,0,0,null);

   // show the new image
   show();
 }

 /**
  * Method to show the picture in a picture frame
  */
 public void show()
 {
    // if there is a current picture frame then use it
   if (pictureFrame != null)
     pictureFrame.updateImageAndShowIt();

   // else create a new picture frame with this picture
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to hide the picture
  */
 public void hide()
 {
   if (pictureFrame != null)
     pictureFrame.setVisible(false);
 }

 /**
  * Method to make this picture visible or not
  * @param flag true if you want it visible else false
  */
 public void setVisible(boolean flag)
 {
   if (flag)
     this.show();
   else
```

```
    this.hide();
 }

 /**
  * Method to open a picture explorer on a copy of this
simple picture
  */
 public void explore()
 {
   // create a copy of the current picture and explore it
   new PictureExplorer(new SimplePicture(this));
 }

 /**
  * Method to force the picture to redraw itself.  This
is very
  * useful after you have changed the pixels in a
picture.
  */
 public void repaint()
 {
   // if there is a picture frame tell it to repaint
   if (pictureFrame != null)
     pictureFrame.repaint();

   // else create a new picture frame
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to load the picture from the passed file name
  * @param fileName the file name to use to load the
picture from
  */
 public void loadOrFail(String fileName) throws
IOException
 {
    // set the current picture's file name
   this.fileName = fileName;

   // set the extension
```

```
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
     this.extension = fileName.substring(posDot + 1);

   // if the current title is null use the file name
   if (title == null)
     title = fileName;

   File file = new File(this.fileName);

   if (!file.canRead())
   {
     // try adding the media path
     file = new
File(FileChooser.getMediaPath(this.fileName));
     if (!file.canRead())
     {
       throw new IOException(this.fileName +
                             " could not be opened. Check
that you specified the path");
     }
   }

   bufferedImage = ImageIO.read(file);
 }


 /**
  * Method to write the contents of the picture to a file
with
  * the passed name without throwing errors
  * @param fileName the name of the file to write the
picture to
  * @return true if success else false
  */
 public boolean load(String fileName)
 {
     try {
         this.loadOrFail(fileName);
         return true;

     } catch (Exception ex) {
```

```
          System.out.println("There was an error trying to
open " + fileName);
          bufferedImage = new BufferedImage(600,200,

BufferedImage.TYPE_INT_RGB);
          addMessage("Couldn't load " + fileName,5,100);
          return false;
      }

 }


 /**
  * Method to load the picture from the passed file name
  * this just calls load(fileName) and is for name
compatibility
  * @param fileName the file name to use to load the
picture from
  * @return true if success else false
  */
 public boolean loadImage(String fileName)
 {
     return load(fileName);
}

 /**
  * Method to draw a message as a string on the buffered
image
  * @param message the message to draw on the buffered
image
  * @param xPos  the leftmost point of the string in x
  * @param yPos  the bottom of the string in y
  */
 public void addMessage(String message, int xPos, int
yPos)
 {
   // get a graphics context to use to draw on the
buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // set the color to white
```

```
//   graphics2d.setPaint(Color.white);
   //modified by baldwin on 12/23/08
   graphics2d.setPaint(messageColor);

   // set the font to Helvetica bold style and size 16
   graphics2d.setFont(new
Font("Helvetica",Font.BOLD,16));

   // draw the message
   graphics2d.drawString(message,xPos,yPos);

 }

 /**
  * Method to draw a string at the given location on the
picture
  * @param text the text to draw
  * @param xPos the left x for the text
  * @param yPos the top y for the text
  */
 public void drawString(String text, int xPos, int yPos)
 {
   addMessage(text,xPos,yPos);
 }

 /**
   * Method to create a new picture by scaling the
current
   * picture by the given x and y factors
   * @param xFactor the amount to scale in x
   * @param yFactor the amount to scale in y
   * @return the resulting picture
   */
  public Picture scale(double xFactor, double yFactor)
  {
    // set up the scale tranform
    AffineTransform scaleTransform = new
AffineTransform();
    scaleTransform.scale(xFactor,yFactor);

    // create a new picture object that is the right size
    Picture result = new Picture((int) (getWidth() *
```

```
xFactor),
                                    (int) (getHeight() *
yFactor));

    // get the graphics 2d object to draw on the result
    Graphics graphics = result.getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    // draw the current image onto the result image
scaled
    g2.drawImage(this.getImage(),scaleTransform,null);

    return result;
  }

  /**
   * Method to create a new picture of the passed width.
   * The aspect ratio of the width and height will stay
   * the same.
   * @param width the desired width
   * @return the resulting picture
   */
  public Picture getPictureWithWidth(int width)
  {
    // set up the scale tranform
    double xFactor = (double) width / this.getWidth();
    Picture result = scale(xFactor,xFactor);
    return result;
  }

  /**
   * Method to create a new picture of the passed height.
   * The aspect ratio of the width and height will stay
   * the same.
   * @param height the desired height
   * @return the resulting picture
   */
  public Picture getPictureWithHeight(int height)
  {
    // set up the scale tranform
    double yFactor = (double) height / this.getHeight();
    Picture result = scale(yFactor,yFactor);
```

```
    return result;
  }

 /**
  * Method to load a picture from a file name and show it
in a picture frame
  * @param fileName the file name to load the picture
from
  * @return true if success else false
  */
 public boolean loadPictureAndShowIt(String fileName)
 {
   boolean result = true;  // the default is that it
worked

   // try to load the picture into the buffered image
from the file name
   result = load(fileName);

   // show the picture in a picture frame
   show();

   return result;
 }

 /**
  * Method to write the contents of the picture to a file
with
  * the passed name
  * @param fileName the name of the file to write the
picture to
  */
 public void writeOrFail(String fileName) throws
IOException
 {
   String extension = this.extension; // the default is
current

   // create the file object
   File file = new File(fileName);
   File fileLoc = file.getParentFile();
```

```
   // canWrite is true only when the file exists already!
(alexr)
   if (!fileLoc.canWrite()) {
       // System.err.println("can't write the file but
trying anyway? ...");
        throw new IOException(fileName +
        " could not be opened. Check to see if you can
write to the directory.");
   }

   // get the extension
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
       extension = fileName.substring(posDot + 1);

   // write the contents of the buffered image to the
file as jpeg
   ImageIO.write(bufferedImage, extension, file);

 }

 /**
  * Method to write the contents of the picture to a file
with
  * the passed name without throwing errors
  * @param fileName the name of the file to write the
picture to
  * @return true if success else false
  */
 public boolean write(String fileName)
 {
     try {
         this.writeOrFail(fileName);
         return true;
     } catch (Exception ex) {
         System.out.println("There was an error trying to
write " + fileName);
         return false;
     }

 }
```

```
 /**
  * Method to set the media path by setting the directory
to use
  * @param directory the directory to use for the media
path
  */
 public static void setMediaPath(String directory) {
   FileChooser.setMediaPath(directory);
 }

 /**
  * Method to get the directory for the media
  * @param fileName the base file name to use
  * @return the full path name by appending
  * the file name to the media directory
  */
 public static String getMediaPath(String fileName) {
   return FileChooser.getMediaPath(fileName);
 }

  /**
   * Method to get the coordinates of the enclosing
rectangle after this
   * transformation is applied to the current picture
   * @return the enclosing rectangle
   */
  public Rectangle2D
getTransformEnclosingRect(AffineTransform trans)
  {
    int width = getWidth();
    int height = getHeight();
    double maxX = width - 1;
    double maxY = height - 1;
    double minX, minY;
    Point2D.Double p1 = new Point2D.Double(0,0);
    Point2D.Double p2 = new Point2D.Double(maxX,0);
    Point2D.Double p3 = new Point2D.Double(maxX,maxY);
    Point2D.Double p4 = new Point2D.Double(0,maxY);
    Point2D.Double result = new Point2D.Double(0,0);
    Rectangle2D.Double rect = null;

    // get the new points and min x and y and max x and y
```

```
    trans.deltaTransform(p1,result);
    minX = result.getX();
    maxX = result.getX();
    minY = result.getY();
    maxY = result.getY();
    trans.deltaTransform(p2,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p3,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p4,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());

    // create the bounding rectangle to return
    rect = new Rectangle2D.Double(minX,minY,maxX - minX +
1, maxY - minY + 1);
    return rect;
  }

 /**
  * Method to return a string with information about this
picture
  * @return a string with information about the picture
  */
 public String toString()
 {
   String output = "Simple Picture, filename " + fileName
+
     " height " + getHeight() + " width " + getWidth();
   return output;
 }

} // end of SimplePicture class
```

-end-

Java3110 Java OOP Panels, Labels, Text Fields, and Buttons
Learn how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the

figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output after one click.
- Figure 3 . Program output after several clicks.

Listings

- Listing 1 . Beginning of the class named Prob05Runner.
- Listing 2 . Instantiate GUI components.
- Listing 3 . Beginning of the constructor.
- Listing 4 . Add the six GUI components to the panel.
- Listing 5 . Get the frame and add the panel to the frame.
- Listing 6 . Set the background to blue and add a turtle.
- Listing 7 . Define, instantiate, and register a listener on the Move button.
- Listing 8 . Action listener to terminate the program.
- Listing 9 . Source code for Prob05.

Preview

In this module, you will learn how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

Just as you did in earlier modules, you will modify the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object. However, because I explained those modifications to the **World** class in earlier modules, I won't repeat the explanation here. You can find a modified version of the World class in Java OOP: Background Color, Text Color, Mouse Clicks, etc.

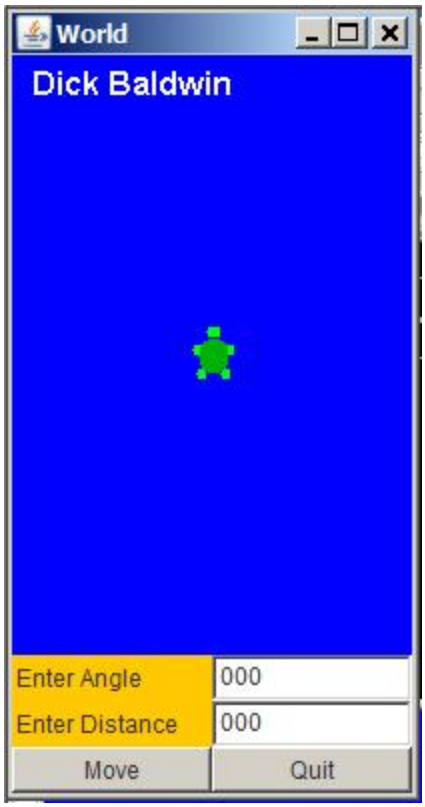
Program specifications

Write a program named **Prob05** that uses the class definition named **Prob05** shown in Listing 9 along with Ericson's media library to produce the graphic output images shown in Figure 1 , Figure 2 , and Figure 3 .

Program output at startup

Figure 1 shows the image that appears on the screen when the program first starts running.

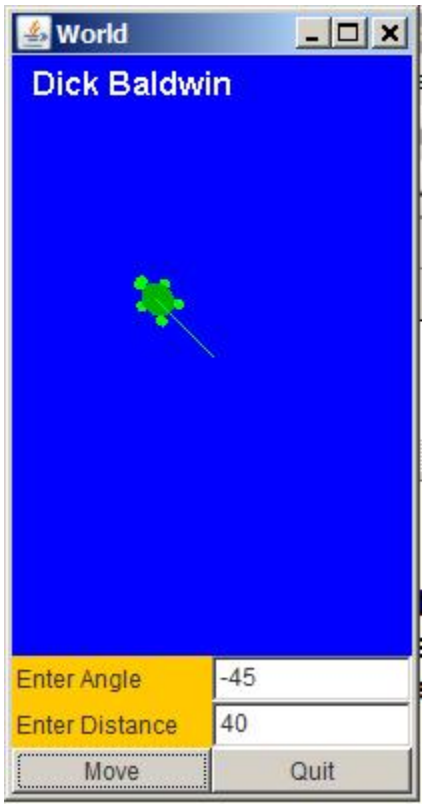
Figure 1 - Program output at startup.



Program output after one click

Figure 2 shows the output image after the user enters the values shown into the two text fields and clicks the **Move** button once.

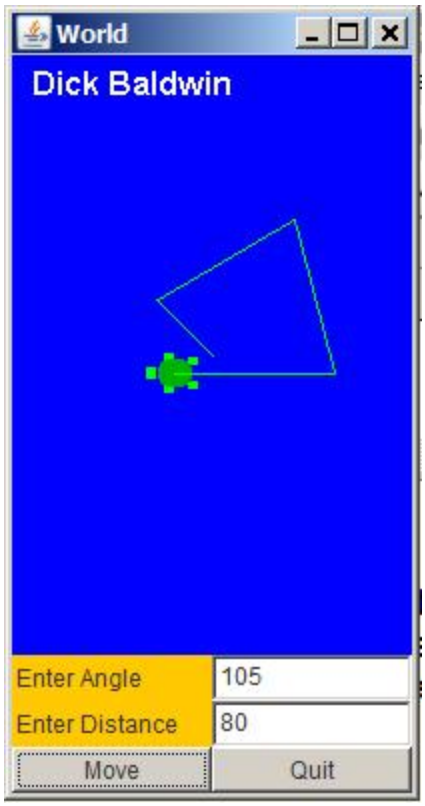
Figure 2 - Program output after one click.



Program output after several clicks

Figure 3 shows the output image after the user enters several different sets of values into the text fields and clicks the **Move** button several times.

Figure 3 - Program output after several clicks.



Labels, text fields, and buttons

As shown in Figure 1 , this program adds two buttons, two labels, and two text fields to form a GUI at the bottom of the **World** object. Although it isn't obvious, those six components are contained in a **Panel** object.

Each time you enter numeric values into the angle and distance fields and then click the **Move** button, the turtle will turn by that angle in degrees and move by that distance in pixels.

Program termination

The program terminates and returns control to the operating system when the user clicks the **Quit** button.

Use AWT components instead of Swing components

The GUI at the bottom of the **World** object is comprised of ***AWT*** components instead of ***Swing*** components.

Required text output

In addition to the output image described above, your program must produce some rather inconsequential text output on the command- line screen.

Discussion and sample code

The class named Prob05

You can view the driver class named **Prob05** at the beginning of the source code in Listing 9 . You are already familiar with the code in the **main** method of that class from earlier modules so I won't waste time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob05Runner** . Once that object is instantiated, the program goes idle waiting for an event to happen. Events happen when the user enters text into the text fields or presses one of the buttons shown in Figure 1 .

The class named Prob05Runner

I will explain this program in fragments. A complete listing of the program is provided in Listing 9 near the end of the module

The class named **Prob05Runner** begins in Listing 1 .

Listing 1 - Beginning of the class named Prob05Runner.

Listing 1 - Beginning of the class named Prob05Runner.

```
class Prob05Runner{
  Turtle turtle = null;
  Picture picture = null;
  World world = new World(200,300);
```

There's nothing new in Listing 1 . You have seen code like this in several earlier modules.

Instantiate GUI components

Listing 2 instantiates the GUI components that are used to construct the GUI at the bottom of Figure 1 .

Listing 2 - Instantiate GUI components.

Listing 2 - Instantiate GUI components.

```
  Panel mainPanel = new Panel();
  Label angleLabel = new Label("Enter Angle");
  TextField angleField = new TextField("000");
  Label distanceLabel = new Label("Enter
Distance");
  TextField distanceField = new
TextField("000");
  Button moveButton = new Button("Move");
  Button quitButton = new Button("Quit");

  int angle = 0;
  int distance = 0;
```

References to the GUI components are saved in instance variables with descriptive names.

GUI component initialization

Except for the **Panel** object, each GUI component is initialized with information that is appropriate to the type of component. Compare the initialization values in Listing 2 with the image in Figure 1 for a better understanding of what I mean by this.

Brief description of the GUI components

You are encouraged to visit the Sun Java documentation and read about the following AWT components:

- **Panel** : A panel provides space in which an application can attach any other components, including other panels.
- **Label** : A Label object is a component for placing text in a container. A label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly.

- **TextField** : A text component that allows for the editing of a single line of text.
- **Button** : This class creates a labeled button. The application can cause some action to happen when the button is pushed.

Figure 1 contains a **Panel** object at the bottom. The panel contains two **Label** objects *(shown as orange)* , two **TextField** objects *(white)* , and two **Button** objects *(gray)* . Those seven objects are instantiated in Listing 2 .

The angle and distance variables

Listing 2 also declares and initializes two instance variables named **angle** and **distance** . *(Note that these two variables would be automatically initialized to zero if I didn't initialize them, but I prefer to initialize them explicitly in order to make the code more self-documenting.)*

Beginning of the constructor

The constructor for the class named **Prob05Runner** begins in Listing 3 .

Listing 3 - Beginning of the constructor.

```
public Prob05Runner(){
  System.out.println("Dick Baldwin");

  mainPanel.setBackground(Color.ORANGE);

  mainPanel.setLayout(new GridLayout(0,2));
```

The constructor begins by displaying my name on the command line screen. This is inconsequential insofar as the overall operation of the program is

concerned.

Set the panel background color to orange

Then Listing 3 sets the background color of the panel to orange. This is what causes the two labels in Figure 1 to appear to be orange. They are actually transparent except for the text. The orange color shows through causing the labels to appear to be orange.

Set the layout to GridLayout

Setting the layout manager controls how the components that are placed in the panel will be arranged. A **GridLayout** causes all components to be the same size arranged in rows and columns.

Overloaded constructors

There are several overloaded versions of the **GridLayout** constructor. For the constructor used in Listing 3 , the parameters specify the number of rows and the number of columns in that order. Specifying the number of rows as 0 and the number of columns as 2 means that the layout manager will accept any number of rows but only two columns.

The order of component placement

The intersections of the rows and columns create ***cells*** . Components are placed in the cells in left to right, top to bottom order as they are added to the panel.

Add the six GUI components to the panel

Listing 4 adds the six GUI components to the panel in the left to right, top to bottom order described above.

Listing 4 - Add the six GUI components to the panel.

```
    mainPanel.add(angleLabel);
    mainPanel.add(angleField);
    mainPanel.add(distanceLabel);
    mainPanel.add(distanceField);
    mainPanel.add(moveButton);
    mainPanel.add(quitButton);
```

At this point, the panel has been populated with GUI components, but the panel itself has not been added to the **JFrame** object that forms the **World** object. That is accomplished in Listing 5 .

Get the frame and add the panel to the frame

Listing 5 gets a reference to the **World** object's frame and adds the panel to the SOUTH location in that frame.

Listing 5 - Get the frame and add the panel to the frame.

```
    JFrame frame = world.getFrame();
    frame.getContentPane().add(

mainPanel,BorderLayout.SOUTH);

    frame.pack();
```

You can surmise from the word SOUTH that the panel is added to the bottom of the frame. To learn more about this, visit the **BorderLayout** class in Sun's Java documentation.

Pack the frame

As you learned in an earlier module, it is very important that you pack the frame at this point. Packing the frame causes the frame to adjust its dimensions in order to accommodate all of the components that have been added to it. In this case, the frame contains a **Picture** object *(placed there when the **World** was constructed)* and a **Panel** object placed there in Listing 5 .

What about the size of the panel?

Exactly how the panel decides what size it needs to be to accommodate the six GUI components is a fairly complicated topic, so I won't go into it here. However, if you do much work developing GUIs, you definitely need to understand the process. I have explained the automatic sizing process in several tutorials on my website.

Set the background to blue and add a turtle

Listing 6 sets the background color of the world to blue and adds a turtle to the world.

Listing 6 - Set the background to blue and add a turtle.

Listing 6 - Set the background to blue and add a turtle.

```
    //Initialize the picture.
    picture = world.getPicture();
    //Set picture background to BLUE
    picture.setAllPixelsToAColor(Color.BLUE);
    //Display the student's name on the
picture.
    picture.addMessage("Dick Baldwin",10,20);
    //Add a turtle to the world. This causes
the
    // world to be repainted.
    turtle = new Turtle(world);
```

There is nothing new in Listing 6 . I have explained code very similar to this code in earlier modules.

Could stop at this point

If we were to stop programming at this point, the program would be executable, and would produce the output shown in Figure 1 when it is run. However it would be completely passive. By that, I mean that entering values into the text fields and clicking the buttons at the bottom of Figure 1 would have no effect. However, the buttons would appear to be active from a visual viewpoint because the animation behavior is built into objects of the **Button** class.

Register listener objects

In order to cause the buttons to impact the behavior of the program, we must instantiate and register listener object on the buttons. I will do that using anonymous classes.

Define, instantiate, and register a listener on the Move button

The code in Listing 7 :

- Defines an anonymous **ActionListener** class.
- Instantiates an anonymous object of that class.
- Registers that object on the **Move** button

Listing 7 - Define, instantiate, and register a listener on the Move button.

```
    moveButton.addActionListener(
      new ActionListener(){

        public void
actionPerformed(ActionEvent e){
          angle =
Integer.parseInt(angleField.getText());
          distance = Integer.parseInt(

distanceField.getText());
          turtle.turn(angle);
          turtle.forward(distance);
        }//end action performed

      }//end newActionListener
    );//end addActionListener
```

The actionPerformed method

As you learned in an earlier module, the code in the **actionPerformed** method is executed each time the user presses the **Move** button.

The **actionPerformed** method in Listing 7 begins by getting the text from each of the text fields, converting the text to type **int** , and saving the **int** values in the variables named **angle** and **distance** .

Then Listing 7 calls the **turn** and **forward** methods of the **Turtle** class to cause the turtle to turn by the specified angle and then move forward by the specified distance.

Action listener to terminate the program

Listing 8 registers an action listener on the **Quit** button, which will cause the program to terminate when the user clicks the button.

Listing 8 - Action listener to terminate the program.

Listing 8 - Action listener to terminate the program.

```
    quitButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){

          System.exit(0);

        }//end action performed
      }//end newActionListener
    );//end addActionListener

  }//end constructor
  //-----------------------------------------
----------//

}//end class Prob05Runner
```

As you can see in Listing 8 , this code causes the **exit** method of the **System** class to be called when the user clicks the **Quit** button. According to the Sun documentation, a call to the **exit** method *"Terminates the currently running Java Virtual Machine."*

The end of the constructor and the end of the class

Listing 8 also shows the end of the constructor for the **Prob05Runner** class and the end of the class.

Run the program

I encourage you to copy the code from Listing 9 . Compile the code and execute it. Experiment with the code, making changes and observing the

results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

What's next?

In the next module, you will learn about:

- Alpha transparency
- A buffered image of type TYPE_INT_ARGB
- The ability to use the **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators,
- The use of the drawImage method of the **Graphics** class.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Panels, Labels, Text Fields, and Buttons
- File: Java3110.htm
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you

should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this lesson is shown in Listing 9 below.

Listing 9 - Source code for Prob05.

```
/*File Prob05 Copyright 2008 R.G.Baldwin
 *Revised 12/31/08
**************************************************
*******/
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.TextField;
import java.awt.Label;
import java.awt.Button;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
```

```
import javax.swing.JFrame;

public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    new Prob05Runner();
  }//end main method
}//end class Prob05
//End program specifications.
/*-----------------------------------------------
------//

***********************************************
*******/

class Prob05Runner{
  Turtle turtle = null;
  Picture picture = null;
  World world = new World(200,300);
  Panel mainPanel = new Panel();
  Label angleLabel = new Label("Enter Angle");
  TextField angleField = new TextField("000");
  Label distanceLabel = new Label("Enter
Distance");
  TextField distanceField = new TextField("000");
  Button moveButton = new Button("Move");
  Button quitButton = new Button("Quit");
  int angle = 0;
  int distance = 0;

  public Prob05Runner(){
    System.out.println("Dick Baldwin");

    //Construct the GUI.
    mainPanel.setBackground(Color.ORANGE);
    mainPanel.setLayout(new GridLayout(0,2));
```

```
    mainPanel.add(angleLabel);
    mainPanel.add(angleField);
    mainPanel.add(distanceLabel);
    mainPanel.add(distanceField);
    mainPanel.add(moveButton);
    mainPanel.add(quitButton);

    //Get a reference to the world frame and add
the GUI
    // to the frame.
    JFrame frame = world.getFrame();
    frame.getContentPane().add(

mainPanel,BorderLayout.SOUTH);


    frame.pack();

    //Initialize the picture.
    picture = world.getPicture();
    //Set picture background to BLUE
    picture.setAllPixelsToAColor(Color.BLUE);
    //Display the student's name on the picture.
    picture.addMessage("Dick Baldwin",10,20);
    //Add a turtle to the world. This causes the
    // world to be repainted.
    turtle = new Turtle(world);

    //-----------------------------------------
------//
    //Register anonymous listeners on the two
buttons.
    moveButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          angle =
```

```
Integer.parseInt(angleField.getText());
          distance = Integer.parseInt(

distanceField.getText());
          turtle.turn(angle);
          turtle.forward(distance);
        }//end action performed
      }//end newActionListener
    );//end addActionListener

    quitButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          System.exit(0);
        }//end action performed
      }//end newActionListener
    );//end addActionListener

  }//end constructor
  //-----------------------------------------
------//

}//end class Prob05Runner
```

-end-

Java3112 Java OOP Using Alpha Transparency with Ericson's Media Library
Learn how to use alpha transparency with Ericson's media library.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- General background information
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Image from file named Prob06a.
- Figure 2 . Image from file named Prob06b.
- Figure 3 . Processed output image.
- Figure 4 . Required text output.

Listings

- Listing 1 . Modification of the SimplePicture class.
- Listing 2 . Beginning of the class named Prob06Runner.
- Listing 3 . The run method.
- Listing 4 . Beginning of the cropAndFlip method.
- Listing 5 . Make the pixels partially transparent.
- Listing 6 . The copyPictureWithCrop method.
- Listing 7 . Complete program listing.

Preview

The primary objective of this module is to incorporate alpha transparency into the use of Ericson's media library.

Two approaches

There are at least two ways to incorporate alpha transparency into Ericson's media library, The easiest way, which is not necessarily the best way, is to make a relatively simple modification to a constructor in Ericson's **SimplePicture** class. That is the approach used in this module.

The second approach

The second approach is more complicated, but does not require the modification of the classes in Ericson's library. That is probably a better approach due simply to the fact that modifications to Ericson's library are not required. However, that approach is not shown in this module.

Outside research

This program may require a significant amount of outside research on the part of the student in order to learn about:

- Alpha transparency
- A buffered image of type **TYPE_INT_ARGB**
- The ability to use Ericson's **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators, and
- The use of the **drawImage** method of the **Graphics** class.

The getBasicPixel and setBasicPixel methods

The program uses the **getBasicPixel** and **setBasicPixel** methods from Ericson's library along with bitwise operations to set the alpha value for all the pixels in a cropped and flipped image of a butterfly to a hexadecimal value of 5F.

Modification to the SimplePicture class

The student must modify the **SimplePicture** class to cause the buffered image used to store the image to be **TYPE_INT_ARGB** instead of **TYPE_INT_RGB** , which is its normal type.

Crop, flip, and set alpha values

Then the student must write a method that will crop and flip an image of a butterfly and set the value of every alpha byte to a hexadecimal value of 5F.

Draw a partially transparent image of a butterfly

Finally, the student must use the standard **drawImage** method of the **Graphics** class to draw the image of the butterfly onto an image of a beach with transparency.

Brief program specifications

Write a program named **Prob06** that uses the class definition for the class named **Prob06** in Listing 7 along with Ericson's media library and the image files named **Prob06a.jpg** and **Prob06b.jpg** to produce the three graphic output images shown in Figure 1 , Figure 2 , and Figure 3 .

Figure 1 - Image from file named Prob06a.

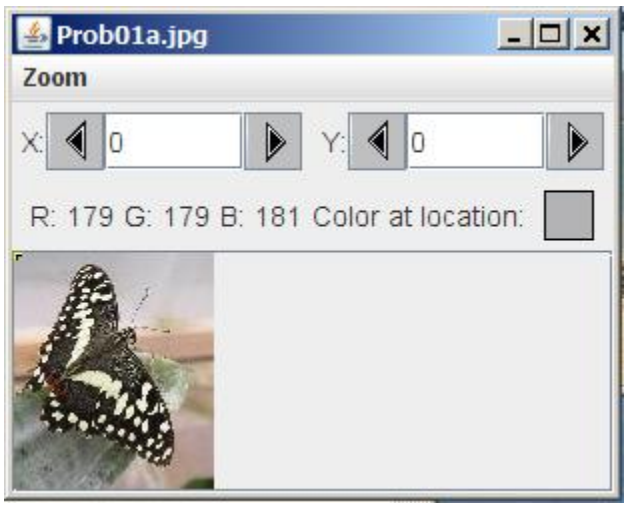


Figure 2 - Image from file named Prob06b.

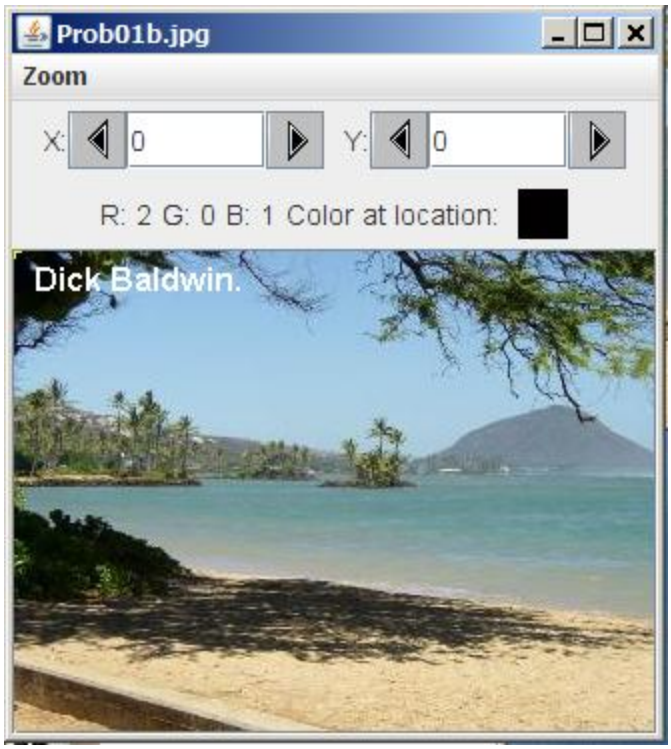
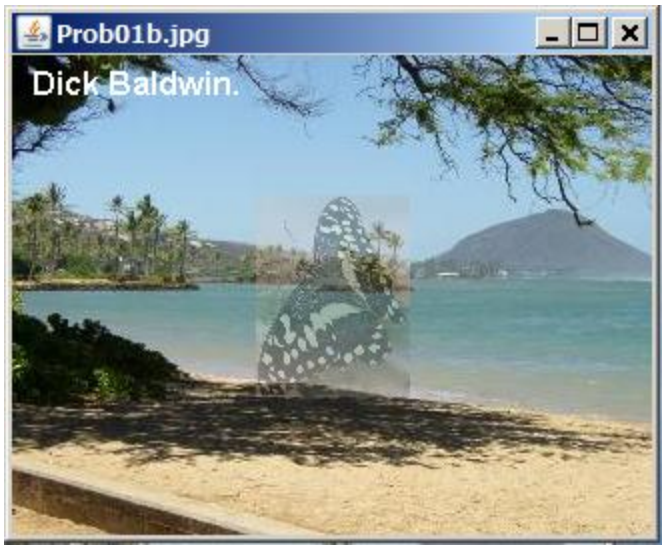


Figure 3 - Processed output image.



Define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob06** given in Listing 7 .

A partially transparent image of a butterfly

Just in case you haven't noticed it, the final image of the beach contains a partially transparent image of a butterfly superimposed and centered on the beach image.

Modification to the SimplePicture class

In order to write this program, you will need to modify the class from Ericson's media library named **SimplePicture** .

Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through.

Transparency

The degree of transparency can range from being completely transparent at one extreme to being totally opaque at the other extreme. In this case, the butterfly image shown in Figure 3 is about 37-percent opaque *(or 63-percent transparent)* .

Outside research

You will probably need to do some outside research in order to write this program. For example, you will need to learn about the following topics and probably some other topics as well:

- Alpha transparency
- BufferedImage objects of TYPE_INT_ARGB
- The representation of a pixel as type int
- Bit manipulation of pixels
- The drawImage method of the Graphics class

Required text output

In addition to the output images described above, your program must produce the text output shown in Figure 4 on the command- line screen.

Figure 4 - Required text output.

```
Dick Baldwin.
Dick Baldwin
Picture, filename Prob06a.jpg height 118 width
100
Picture, filename Prob06b.jpg height 240 width
320
Picture, filename None height 101 width 77
```

You must substitute your name for my name wherever my name appears both in the images and on the command-line screen.

General background information

The image in a **SimplePicture** object is stored in an object of the **BufferedImage** class, which is a class in the standard Sun Java library.

Image data formats

An examination of the documentation for the **BufferedImage** class shows that the red, green, blue, and alpha values for each pixel can be formatted in about fourteen different ways in an object of the **BufferedImage** class.

No alpha data

Some of those formats, including the way that information is stored in a **SimplePicture** object, don't include an alpha value.

Modification of the SimplePicture class

One way to modify the **SimplePicture** class to force it to accommodate alpha transparency data is to modify one of the constructors for the **SimplePicture** class as shown in Listing 1 . Note that **BufferedImage.TYPE_INT_RGB** was replaced by

BufferedImage.TYPE_INT_ARGB in Listing 1. *(There are probably other ways that you can modify the class to achieve the same result as well.)*

Listing 1 - Modification of the SimplePicture class.

Listing 1 - Modification of the SimplePicture class.

```
 /**
  * A constructor that takes the width and
height desired
  * for a picture and creates a buffered image
of that
  * size.  This constructor doesn't show the
picture.
  */
 public  SimplePicture(int width, int height){
//Disable the following statement
//   bufferedImage = new BufferedImage(
//            width, height,
BufferedImage.TYPE_INT_RGB);

   //Modify constructor to support alpha
transparency.
   System.out.println("Dick Baldwin");
   bufferedImage = new BufferedImage(
             width, height,
BufferedImage.TYPE_INT_ARGB);

   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }//end constructor
```

Future Picture objects will accommodate alpha transparency

Having made this modification, future objects instantiated from the **SimplePicture** class using this constructor will accommodate alpha transparency. *(The **SimplePicture** class is the superclass of the **Picture** class.)*

Display the student's name

Note that the constructor in Listing 1 is also modified to cause it to display the student's name, which is a requirement of the program.

No complete listing of SimplePicture provided

Because of the simplicity of this modification, a complete listing of the modified **SimplePicture** class will not be provided in this module.

Discussion and sample code

The class named Prob06

You can view the driver class named **Prob06** at the beginning of the source code in Listing 7 . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob06Runner** and calls the **run** method on that object. When the **run** method returns, the code in the **main** method displays some information about the three images and terminates.

(Because there are images on the screen, the program does not actually terminate until the user forces it to terminate.)

The class named Prob06Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 7 near the end of the module

The class named **Prob06Runner** begins in Listing 2 , which shows the constructor for the class.

Listing 2 - Beginning of the class named Prob06Runner.

```
class Prob06Runner{

  public Prob06Runner(){//constructor
    System.out.println("Dick Baldwin.");
  }//end constructor
```

The constructor simply displays the student's name to satisfy one of the requirements of the program.

The run method

The run method, which is called from the **main** method in Listing 7 , is shown in its entirety in Listing 3 .

Listing 3 - The run method.

Listing 3 - The run method.

```
  public Picture[] run(){
    //Insert executable code here
    Picture picA = new Picture("Prob06a.jpg");
    picA.explore();
    Picture picB = new Picture("Prob06b.jpg");
    picB.addMessage("Dick Baldwin.",10,20);
    picB.explore();

    Picture picC =
cropAndFlip(picA,4,5,80,105);

    copyPictureWithCrop(picC,picB,122,70);

    picB.show();

    Picture[] output = {picA,picB,picC};
    return output;
  }//end run
```

New material

The only thing in Listing 3 that is new to this module is the pair of calls to the following methods. I will explain these methods in the paragraphs that follow:

- **cropAndFlip**
- **copyPictureWithCrop**

Beginning of the cropAndFlip method

The **cropAndFlip** method begins in Listing 4 . This method receives an incoming reference to a **Picture** object. It crops the picture to a set of specified coordinate values and flips it around a vertical line at its center.

Listing 4 - Beginning of the cropAndFlip method.

```
  private Picture cropAndFlip(
                  Picture pic,int x1,int y1,int
x2,int y2){
    Picture output = new Picture(x2-x1+1,y2-
y1+1);

    int width = output.getWidth();
    Pixel pixel = null;
    Color color = null;
    for(int col = x1;col < (x2+1);col++){
      for(int row = y1;row < (y2+1);row++){
        color =
pic.getPixel(col,row).getColor();
        pixel = output.getPixel(width-col+x1-
1,row-y1);
        pixel.setColor(color);
      }//end inner loop
    }//end outer loop
```

Receives a reference to the butterfly image

Note from Listing 3 that the **cropAndFlip** method receives a reference to the **Picture** object of the butterfly that is displayed in Figure 1 .

Also note that the butterfly in Figure 1 is facing toward the right while the butterfly in the output image in Figure 3 has been cropped to a smaller size and is facing toward the left.

Crop and flip is not new

The capability to crop and flip an image is not new to this module. However, the **cropAndFlip** method also makes the image partially

transparent as shown in Figure 3 . That capability is new to this module. I will explain how that is done shortly.

A call to the modified SimplePicture constructor

Although there is nothing new in the code in Listing 4 , it is important to note that the first statement in Listing 4 causes the **SimplePicture** constructor that was modified in Listing 1 to be called.

As a result, the **Picture** object referred to by the reference variable named **output** in Listing 4 will accommodate alpha transparency data.

Make the pixels partially transparent

The code in Listing 5 uses a pair of nested **for** loops to iterate through all of the pixels in the picture referred to by **output** and modify each pixel.

The four statements in the body of the inner loop in Listing 5 cause the current pixel to become partially transparent.

Listing 5 - Make the pixels partially transparent.

Listing 5 - Make the pixels partially transparent.

```
    width = output.getWidth();
    int height = output.getHeight();
    pixel = null;
    color = null;
    for(int col = 0;col < width;col++){
      for(int row = 0;row < height;row++){

        int basicPixel =
output.getBasicPixel(col,row);

        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | 0x5F000000;


output.setBasicPixel(col,row,basicPixel);

      }//end inner loop
    }//end outer loop

    return output;
  }//end crop and flip
```

The getBasicPixel method

According to Ericson's documentation, the **getBasicPixel** method will *"return the pixel value as an int for the given x and y location."* In other words, a call to the **getBasicPixel** method will return an **int** value containing the red, green, blue, and alpha values for the pixel at the specified location.

A bitwise AND operation

Listing 5 uses a bitwise ***AND*** operation *(note the single ampersand)* to force the eight most significant bits *(the alpha byte)* in the **int** representation of the current pixel to zero while preserving the bit values stored in the least significant 24 bits.

A bitwise OR operation

Then Listing 5 uses a bitwise ***OR*** operation (|) to store the hexadecimal value 5F in the eight most significant bits *(the alpha byte)* without changing the values stored in the 24 least significant bits.

The alpha byte

The value of the alpha byte can range from 0 to 255. When rendered using a mechanism that supports alpha transparency, an alpha value of zero causes the pixel to be totally transparent.

Similarly, an alpha value of 255 causes the pixel to be totally opaque.

Values between zero and 255 cause the pixel to be rendered as partially opaque or partially transparent, whichever terminology you prefer.

Thirty-seven percent opaque

If I did the arithmetic correctly, a hexadecimal value of 5F represents a decimal value of 95. Therefore, this value will cause the pixel to be about 37-percent opaque *(or 63-percent transparent)* .

The setBasicPixel method

As the name implies, the **setBasicPixel** method can be used to *"set the value of a pixel in the picture from an int."*

Therefore, the last statement in the body of the inner loop in Listing 5 replaces the value of the current pixel with the modified value containing a value of 95 in the alpha byte.

The end of the cropAndFlip method

When the pair of nested **for** loops in Listing 5 terminates, the **cropAndFlip** method returns control to the **run** method in Listing 3 , returning a copy of the reference from the variable named **output** *(see Listing 4)* in the process.

Save the Picture object's reference

The returned reference is stored in the reference variable named **picC** in Listing 3 .

At this point, **picC** contains a reference to a butterfly image that has been cropped, flipped, and formatted into a buffered image that contains alpha transparency information.

Call the copyPictureWithCrop method

Listing 3 immediately calls the **copyPictureWithCrop** method passing copies of the references stored in **picC** and **picB** along with a pair of integer coordinate values.

The copyPictureWithCrop method

The **copyPictureWithCrop** method is shown in its entirety in Listing 6 .

Listing 6 - The copyPictureWithCrop method.

Listing 6 - The copyPictureWithCrop method.

```
  private void copyPictureWithCrop(
           Picture source,Picture dest,int
xOff,
                                       int
yOff){

      Graphics destGraphics =
dest.getGraphics();
      Image sourceImage = source.getImage();
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end copyPictureWithCrop method
}//end class Prob06Runner
```

The purpose of the **copyPictureWithCrop** method is to copy a source picture onto a destination picture with an offset on each axis.

An exercise for the student

I won't attempt to explain the code in Listing 6 in this module. Instead, I will simply suggest that you go to Google and search for the following or similar keywords:

baldwin java drawImage

You will find many tutorials that I have written that deal with topics in this area.

Modify the destination pixel colors

I will tell you that the use of the **drawImage** method in Listing 6 modifies the destination picture in such a way that the color of each pixel in the resulting image is a combination of the colors in the original destination image and the corresponding pixel in the source image.

An illusion of transparency

If a source pixel is totally transparent, it has no effect on the color of the destination pixel.

If the source pixel is totally opaque, the color of the destination pixel is changed to the color of the source pixel.

For alpha values between these two extremes, the final color of the destination pixel produces the illusion of a partially transparent image in front of the original destination image.

Termination of the copyPictureWithCrop method

When the **copyPictureWithCrop** method terminates in Listing 6 , control returns to the run method in Listing 3 .

Listing 3 calls the **show** method to display the image in the now-modified **Picture** object referred to by **picB** , as shown in Figure 3 .

Return a reference to an array object

Then the **run** method encapsulates references to each of the three images in an array object and returns control to the **main** method in Listing 7 , returning a copy of the array object's reference in the process.

The **main** method in Listing 7 displays information about each of the three **Picture** objects, producing the output shown in Figure 4 . Then the main method terminates.

Images don't go away immediately

Because there are images belonging to the program still on the screen, the program doesn't return control to the operating system. It will simply wait

until it is forced to terminate by the user before returning control to the operating system.

Clicking the X-buttons in the upper-right corners of the images will simply hide the frames and won't terminate the program. Some extra work is required to deal with this issue.

Run the program

I encourage you to copy the code from Listing 7 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned about:

- Alpha transparency
- A buffered image of type TYPE_INT_ARGB
- The ability to use the **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators,
- The use of the **drawImage** method of the **Graphics** class.

You modified the **SimplePicture** class to cause the buffered image used to store the image to be TYPE_INT_ARGB instead of TYPE_INT_RGB, which is its normal type.

You wrote a method that cropped and flipped an image of a butterfly.

You used the **getBasicPixel** and **setBasicPixel** methods from Ericson's library along with bitwise operations to set the alpha value for all the pixels in the cropped and flipped image of the butterfly to a hexadecimal value of 5F.

Finally, you used the standard **drawImage** method of the **Graphics** class to draw the image of the butterfly onto an image of a beach with transparency.

What's next?

In the next module, you will learn how to use a slider to continuously change the opacity of an image and to draw that modified image onto a background image.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Using Alpha Transparency with Ericson's Media Library
- File: Java3112.htm
- Published: 05/13/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available

on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is shown in Listing 7 below.

Listing 7 - Complete program listing.

```
/*File Prob06 Copyright 2008 R.G.Baldwin
Revised 12/31/08
*************************************************
*******/
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;

public class Prob06{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    Picture[] pictures = new Prob06Runner().run();
    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob06
//=============================================
======//

class Prob06Runner{

  public Prob06Runner(){//constructor
```

```
    System.out.println("Dick Baldwin.");
  }//end constructor
  //-----------------------------------------
------//
  public Picture[] run(){
    //Insert executable code here
    Picture picA = new Picture("Prob06a.jpg");
    picA.explore();
    Picture picB = new Picture("Prob06b.jpg");
    picB.addMessage("Dick Baldwin.",10,20);
    picB.explore();

    Picture picC = cropAndFlip(picA,4,5,80,105);
    copyPictureWithCrop(picC,picB,122,70);

    picB.show();

    Picture[] output = {picA,picB,picC};
    return output;
  }//end run
  //-----------------------------------------
------//

  //Crops a picture to the specified coordinate
values and
  // flips it around a vertical line at its
center.
  //Also makes it partially transparent
  private Picture cropAndFlip(
                  Picture pic,int x1,int y1,int
x2,int y2){
    Picture output = new Picture(x2-x1+1,y2-y1+1);

    int width = output.getWidth();
    Pixel pixel = null;
    Color color = null;
    for(int col = x1;col < (x2+1);col++){
```

```
      for(int row = y1;row < (y2+1);row++){
        color = pic.getPixel(col,row).getColor();
        pixel = output.getPixel(width-col+x1-
1,row-y1);
        pixel.setColor(color);
      }//end inner loop
    }//end outer loop

    width = output.getWidth();
    int height = output.getHeight();
    pixel = null;
    color = null;
    for(int col = 0;col < width;col++){
      for(int row = 0;row < height;row++){

        int basicPixel =
output.getBasicPixel(col,row);
        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | 0x5F000000;
        output.setBasicPixel(col,row,basicPixel);
      }//end inner loop
    }//end outer loop


    return output;
  }//end crop and flip
  //-----------------------------------------
------//

  //Copies the source picture onto the destination
  // picture with an offset on both axes.
  private void copyPictureWithCrop(
           Picture source,Picture dest,int xOff,
                                       int yOff){

      Graphics destGraphics = dest.getGraphics();
      Image sourceImage = source.getImage();
```

```
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end copyPictureWithCrop method
}//end class Prob06Runner
```

-end-

Java3114 Java OOP Controlling Opacity with a Slider
Learn how to use a slider to continuously change the opacity of an image and to draw that image onto a background image.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- General background information
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Screen output at startup.
- Figure 2 . Twenty-percent opacity.
- Figure 3 . Eighty-percent opacity.

Listings

- Listing 1 . Modification of the SimplePicture class.
- Listing 2 . Beginning of the class named Prob07Runner.
- Listing 3 . Beginning of the constructor.
- Listing 4 . Display the initial image.
- Listing 5 . Display the butterfly at 50-percent opacity.
- Listing 6 . The setOpacity method.
- Listing 7 . The drawPictureOnPicture method.
- Listing 8 . Begin the registration of an event handler on the slider.
- Listing 9 . Draw the butterfly and repaint.
- Listing 10 . Complete program listing.

Preview

The primary objective of this module is to illustrate how to use a slider to continuously change the opacity of an image and to draw that image onto a background image.

Two approaches

This module builds on an earlier module involving transparency. In that module, you learned that there are at least two ways to incorporate alpha transparency into Ericson's media library, The easiest way, which is not necessarily the best way, is to make a relatively simple modification to a constructor in Ericson's **SimplePicture** class. That is the approach used in this module.

The second approach

The second approach is more complicated, but does not require the modification of the classes in Ericson's library. That is probably a better approach due simply to the fact that modifications to Ericson's library are not required. However, that approach is not shown in this module.

Outside research

As with the earlier module, the program that I will explain in this module may require a significant amount of outside research on the part of the student in order to learn about:

- Alpha transparency
- A buffered image of type **TYPE_INT_ARGB**
- The ability to use Ericson's **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND, OR, and left-shift operators.
- The use of the **drawImage** method of the **Graphics** class.

Modification to the SimplePicture class

The student must modify the **SimplePicture** class to cause the buffered image used to store the image to be **TYPE_INT_ARGB** instead of **TYPE_INT_RGB** , which is its normal type.

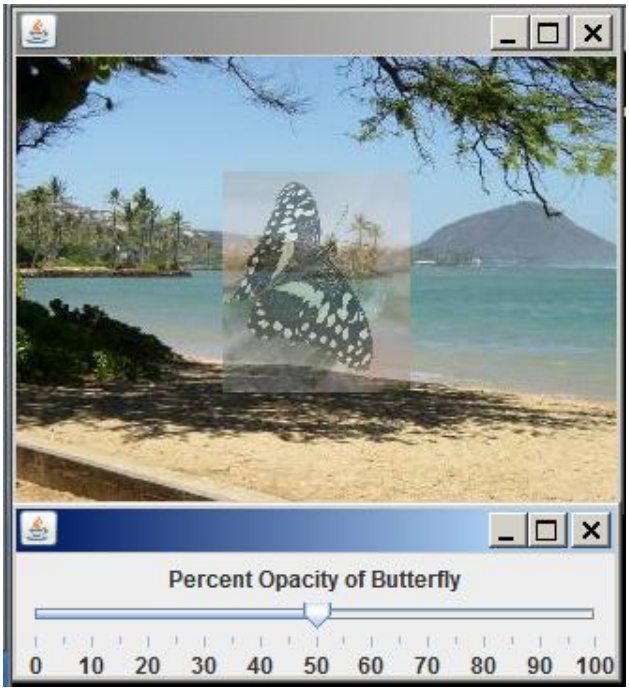
Generally speaking, this program:

- Instantiates a new visual object that extends the **JFrame** class and contains a **JSlider** object.
- Instantiates **Picture** objects from two image files *(beach and butterfly)* along with some blank **Picture** objects of the same size.
- Defines a method named **setOpacity** that can be called to set the opacity of every pixel in a picture to a specified value.
- Defines a method named **drawPictureOnPicture** that can be called to draw one picture onto another picture.
- Registers a **ChangeEvent** handler on the slider to:
    - Extract a percent-opacity value from the slider based on the position of the thumb.
    - Apply that opacity value to the butterfly image.
    - Draw the modified butterfly image on the beach image and display it.

Brief program specifications

Write a program named **Prob07** that uses the class definition for the class named **Prob07** in Listing 10 along with Ericson's media library and the image files named Prob07a.jpg and Prob07b.jpg to produce the two output images shown in Figure 1 .

Figure 1 - Screen output at startup.



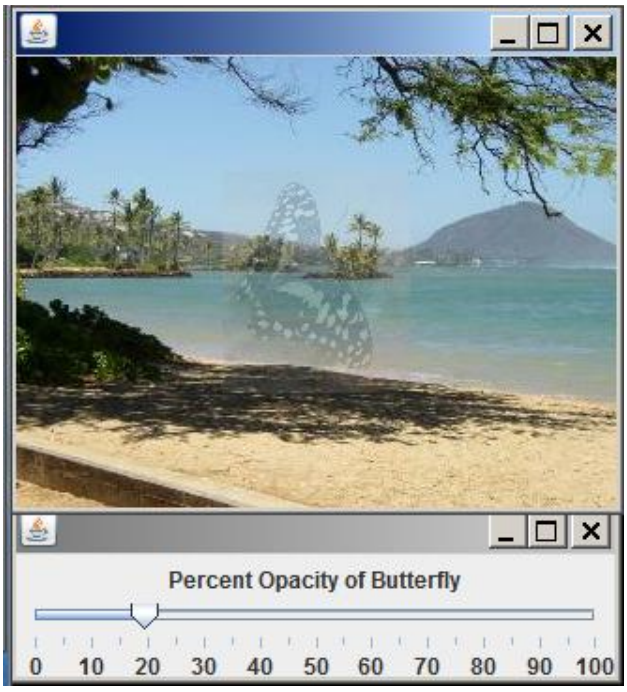
Two output images

Note that Figure 1 actually consists of two output images, one positioned below the other.

Move the thumb to the left

When you move the thumb on the slider to the left, the butterfly becomes less opaque *(more transparent)* as shown in Figure 2 with total transparency at the extreme left end of the slider.

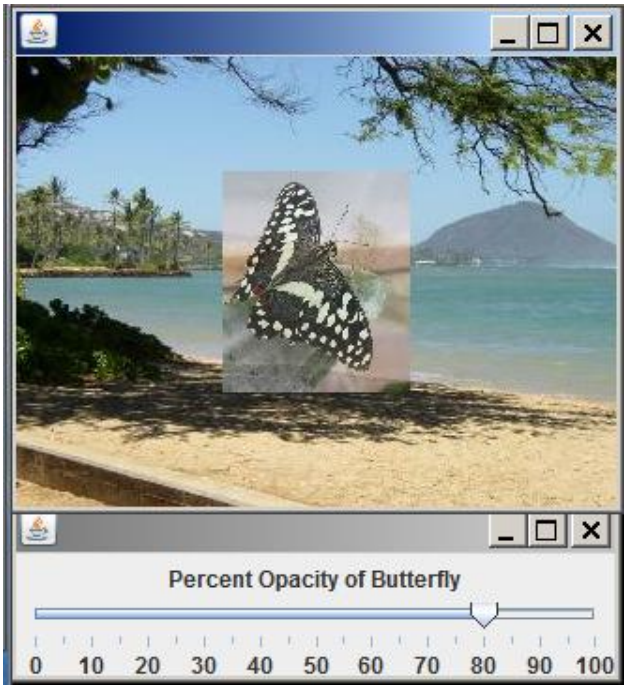
Figure 2 - Twenty-percent opacity.



Move the thumb to the right

When you move the thumb on the slider to the right, the butterfly becomes more opaque *(less transparent)* as shown in Figure 3 with total opacity at the extreme right end of the slider.

Figure 3 - Eighty-percent opacity.



Define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob07** given in Listing 10 .

General background information

The image in a **SimplePicture** object is stored in an object of the **BufferedImage** class, which is a class in the standard Sun Java library.

Image data formats

An examination of the documentation for the **BufferedImage** class shows that the red, green, blue, and alpha values for each pixel can be formatted in about fourteen different ways in an object of the **BufferedImage** class.

No alpha data

Some of those formats, including the way that information is stored in a **SimplePicture** object, don't include an alpha value.

Modification of the SimplePicture class

One way to modify the **SimplePicture** class to force it to accommodate alpha transparency data is to modify one of the constructors for the **SimplePicture** class as shown in Listing 1 . Note the change indicated by comments in Listing 1 . *(There are probably other ways that you can modify the class to achieve the same result as well.)*

Listing 1 - Modification of the SimplePicture class.

Listing 1 - Modification of the SimplePicture class.

```
 /**
  * A constructor that takes the width and height
desired
  * for a picture and creates a buffered image of
that
  * size.  This constructor doesn't show the
picture.
  */
 public  SimplePicture(int width, int height){
//Disable the following statement
//   bufferedImage = new BufferedImage(
//            width, height,
BufferedImage.TYPE_INT_RGB);

   //Modify constructor to support alpha
transparency.
   bufferedImage = new BufferedImage(
             width, height,
BufferedImage.TYPE_INT_ARGB);

   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }//end constructor
```

Future Picture objects will accommodate alpha transparency

Having made this modification, future objects instantiated from the **SimplePicture** class using this constructor will accommodate alpha transparency. *(The **SimplePicture** class is the superclass of the **Picture** class.)*

No complete listing of SimplePicture provided

Because of the simplicity of this modification, a complete listing of the modified **SimplePicture** class will not be provided in this module.

Discussion and sample code

The class named Prob07

You can view the driver class named **Prob07** at the beginning of the source code in Listing 10 . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob07Runner** and calls the **run** method on that object. When the **run** method returns, the GUI shown in Figure 1 has been displayed on the screen.

At that point, the program simply goes into an idle state and waits for the user to take some action that causes an event to be fired. When an event is fired, it is handled and the program goes idle again waiting for another event.

(Because there are images on the screen, the program does not actually terminate until the user forces it to terminate.)

The class named Prob07Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 10 near the end of the module.

Beginning of the class named Prob07Runner

The class named **Prob07Runner** begins in Listing 2 .

Listing 2 - Beginning of the class named Prob07Runner.

```
class Prob07Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider();

  private Picture background = new
Picture("Prob07b.jpg");
  private Picture butterfly = new
Picture("Prob07a.jpg");


  private int backgroundWidth =
background.getWidth();
  private int backgroundHeight =
background.getHeight();
  private int butterflyWidth =
butterfly.getWidth();
  private int butterflyHeight =
butterfly.getHeight();

  private Picture display =
            new
Picture(backgroundWidth,backgroundHeight);
  private Picture tempPicture =
              new
Picture(butterflyWidth,butterflyHeight);

  private Image image = null;
  private Graphics graphics = null;
```

Class extends JFrame

Note that this class extends **JFrame** . An object of this class forms the lower part of the image shown in Figure 1 that contains the slider.

The code in Listing 1 is straightforward and shouldn't require an explanation.

When Listing 2 finishes executing...

When the code in Listing 2 has finished executing, four new **Picture** objects have been instantiated and referred to by the following reference variables:

- **background** - The beach scene shown in the background in Figure 1 .
- **butterfly** - Contains an opaque image of the butterfly shown in Figure 1 .
- **display** - Empty picture the same size as the beach scene.
- **tempPicture** - Empty picture the same size as the butterfly.

In addition, a pair of working variables named **image** and **graphics** of the types **Image** and **Graphics** have been declared.

Finally, when the code in Listing 2 has finished executing, two new **JPanel** objects and one new **JSlider** object have been instantiated and referred to by the variables named **mainPanel** , **titlePanel** , and **slider** .

Beginning of the constructor

The beginning of the constructor is shown in Listing 3 .

Listing 3 - Beginning of the constructor.

Listing 3 - Beginning of the constructor.

```
  public Prob07Runner(){//constructor
    //Do some initial setup.

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                          "Percent Opacity of
Butterfly"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(backgroundWidth + 7,97);
    setLocation(0,backgroundHeight + 25);
    setVisible(true);
```

Although it may be necessary for you to go to Sun's Java documentation to learn about the detailed behavior of some of the methods that are called in Listing 3 , the code in Listing 3 is straightforward and should not require further explanation.

Display the initial background image

Listing 4 displays the initial background image.

Instantiating and destroying a lot of new **Picture** objects as the user moves the slider to change the opacity would be very inefficient. To avoid this inefficiency,

this program gets images from existing **Picture** objects and draws them on existing **Picture** objects without modifying the originals.

Listing 4 - Display the initial image.

```
    graphics = display.getGraphics();

graphics.drawImage(background.getImage(),0,0,null);
```

Display the butterfly at 50-percent opacity

Listing 5 calls the **setOpacity** and **drawPictureOnPicture** methods to set the opacity of the butterfly and draw it onto the display with 50-percent opacity. The image of the butterfly is centered on the background.

Listing 5 - Display the butterfly at 50-percent opacity.

Listing 5 - Display the butterfly at 50-percent opacity.

```
    butterfly = setOpacity(butterfly,50);
    drawPictureOnPicture(
                  butterfly,
                  display,
                  backgroundWidth/2 -
butterflyWidth/2,
                  backgroundHeight/2 -
butterflyHeight/2);
    display.show();
```

Put the constructor on hold

At this point, I will put the discussion of the constructor on hold and explain the **setOpacity** and **drawPictureOnPicture** methods.

The setOpacity method

The **setOpacity** method is shown in its entirety in Listing 6 .

Listing 6 - The setOpacity method.

Listing 6 - The setOpacity method.

```
  private Picture setOpacity(
                  Picture pic,double
percentOpacity){

    int opacity = (int)(255*percentOpacity/100);
    int opacityMask = opacity << 24;

    for(int col = 0;col < butterflyWidth;col++){
      for(int row = 0;row < butterflyHeight;row++)
{
        //Get the pixel in basic int format.
        int basicPixel =
pic.getBasicPixel(col,row);

        //Set the alpha value for the pixel.
        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | opacityMask;

        //Set the modified pixel into tempPicture.

tempPicture.setBasicPixel(col,row,basicPixel);
      }//end inner loop
    }//end outer loop

    return tempPicture;

  }//end setOpacity
```

This method copies an incoming picture into an existing temporary picture, setting the alpha value for every pixel to a specified value in the process. Then it

returns the modified picture object's reference where it is saved in the reference variable named **butterfly** in Listing 5 ,

A bitwise left-shift operation

The only thing in Listing 6 that is new to this module is the use of a bitwise left-shift operation.

A 24-bit left shift

Listing 6 converts the incoming **percentOpacity** value to an integer value ranging from 0 to 255. This value resides in the least significant eight bits of an **int** variable named **opacity** .

Then Listing 6 applies the bitwise left-shift operator *(two left angle brackets)* to shift those eight bits into the eight most significant bits and stores the result in another **int** variable named **opacityMask** .

Apply the opacityMask to the pixels

A pair of nested **for** loops is used to set the alpha value of every pixel to the value of **opacityMask** using an overall bit-masking methodology that I explained in an earlier module.

The drawPictureOnPicture method

After the alpha value for every pixel in the butterfly image has been set to the specified opacity, Listing 5 calls the method named **drawPictureOnPicture** to draw the modified butterfly image on the beach scene as shown in Figure 1 .

The **drawPictureOnPicture** method is shown in its entirety in Listing 7 .

Listing 7 - The drawPictureOnPicture method.

Listing 7 - The drawPictureOnPicture method.

```
  private void drawPictureOnPicture(
                      Picture source,Picture
dest,int xOff,

int yOff){

      Graphics destGraphics = dest.getGraphics();
      Image sourceImage = source.getImage();
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end drawPictureOnPicture method
```

This method draws the source picture onto the destination picture with an offset on both axes. There is nothing in Listing 7 that I haven't explained in an earlier module.

Return to the explanation of the constructor

You are already familiar with the use of anonymous inner classes to create and register listener objects on Java source objects. The slider is a source object.

Listing 8 begins the registration of an anonymous **ChangeEvent** listener on the slider.

Listing 8 - Begin the registration of an event handler on the slider.

Listing 8 - Begin the registration of an event handler on the slider.

```
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          //Draw a new copy of the background on
the
          // display.
          graphics = display.getGraphics();
          graphics.drawImage(

background.getImage(),0,0,null);
```

Restore the background image

Each time the slider fires a **ChangeEvent** , this event handler draws a new background image on the display. This erases what was previously drawn there, restoring a pristine image of the beach scene.

Draw a partially opaque butterfly image on the background

Then it uses the current value of the slider to set the opacity of the butterfly image and draws it centered on the display on top of the background image.

A series of events

The slider fires a series of **ChangeEvents** as the user moves the thumb on the slider. Listing 8 begins the definition of the event handler method named **stateChanged** , which is registered on the slider. This method is called each time the slider fires a **ChangeEvent** .

Listing 8 draws a new copy of the beach background image on the **Picture** object referred to by the reference variable named **background** . This image replaces the image that was previously drawn there.

Draw the butterfly and repaint

Listing 9 calls the **setOpacity** and **drawPictureOnPicture** methods to:

- Set the opacity of the butterfly to the value currently represented by the position of the thumb on the slider. This is the value returned by the slider's **getValue** method.
- Draw the butterfly image on the background image.

Listing 9 - Draw the butterfly and repaint.

```
          //Set the opacity of butterfly and copy
it onto
          // the display. Then repaint the
display.
          butterfly =

setOpacity(butterfly,slider.getValue());
          drawPictureOnPicture(
                  butterfly,
                  display,
                  backgroundWidth/2 -
butterflyWidth/2,
                  backgroundHeight/2 -
butterflyHeight/2);

          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------
------//
  }//end constructor
```

Repaint the image

Then Listing 9 calls the **repaint** method to cause the modified image to be rendered onto the computer screen.

The end of the program

Listing 9 also signals the end of the constructor, the end of the class named **Prob07Runner** , and the end of the program.

Run the program

I encourage you to copy the code from Listing 10 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to use a slider to continuously change the opacity of an image and draw that image onto a background image.

What's next?

In the next module, you will learn how to use a slider to continuously change the threshold detection level of an edge detector and to draw the edge-detected image on the screen.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Controlling Opacity with a Slider
- File: Java3114.htm
- Published: 05/13/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 below.

Listing 10 - Complete program listing.

```
/*File Prob07 Copyright 2008 R.G.Baldwin
*************************************************
***/
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
```

```
public class Prob07{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob07Runner();
  }//end main method
}//end class Prob07
//=================================================
==//

class Prob07Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider();

  private Picture background = new
Picture("Prob07b.jpg");
  private Picture butterfly = new
Picture("Prob07a.jpg");


  private int backgroundWidth = background.getWidth();
  private int backgroundHeight =
background.getHeight();
  private int butterflyWidth = butterfly.getWidth();
  private int butterflyHeight = butterfly.getHeight();

  private Picture display =
            new
Picture(backgroundWidth,backgroundHeight);
  private Picture tempPicture =
              new
Picture(butterflyWidth,butterflyHeight);

  private Image image = null;
  private Graphics graphics = null;

  public Prob07Runner(){//constructor
    //Do some initial setup.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                          "Percent Opacity of
Butterfly"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(backgroundWidth + 7,97);
    setLocation(0,backgroundHeight + 25);
    setVisible(true);

    //Draw and display the initial image with 50-
percent
    // opacity. In order to avoid instantiating and
    // destroying a lot of Picture objects, the
    // procedure is to simply get images from existing
    // picture objects and draw them on other existing
    // picture objects.
    graphics = display.getGraphics();

graphics.drawImage(background.getImage(),0,0,null);

    //Set the opacity of butterfly and draw it onto
the
    // display. In this case, the opacity is set to
    // 50-percent. The image of the butterfly is
centered
    // on the background.
    butterfly = setOpacity(butterfly,50);
    drawPictureOnPicture(
                  butterfly,
                  display,
```

```
                    backgroundWidth/2 -
butterflyWidth/2,
                    backgroundHeight/2 -
butterflyHeight/2);
    display.show();
    //-----------------------------------------------
--//
    //Register an anonymous listener object on the
slider.
    //Each time the slider fires a ChangeEvent, this
event
    // handler draws a new background image on the
    // display. This erases what was previously drawn
    // there. Then it uses the current value of the
slider
    // to set the opacity of the butterfly image and
    // draws it on the display on top of the
background
    // image. It is centered on the background image.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          //Draw a new copy of the background on the
          // display.
          graphics = display.getGraphics();
          graphics.drawImage(

background.getImage(),0,0,null);

          //Set the opacity of butterfly and copy it
onto
          // the display. Then repaint the display.
          butterfly =

setOpacity(butterfly,slider.getValue());
          drawPictureOnPicture(
                  butterfly,
                  display,
                  backgroundWidth/2 -
butterflyWidth/2,
```

```
                  backgroundHeight/2 -
butterflyHeight/2);
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------------
--//
  }//end constructor
  //-------------------------------------------------
--//


  //This method copies an incoming picture into an
  // existing temporary picture, setting the alpha
value
  // for every pixel to a specified value. Then it
returns
  // the modified temporary picture object.
  private Picture setOpacity(
                       Picture pic,double
percentOpacity){

    int opacity = (int)(255*percentOpacity/100);
    int opacityMask = opacity << 24;

    for(int col = 0;col < butterflyWidth;col++){
      for(int row = 0;row < butterflyHeight;row++){
        //Get the pixel in basic int format.
        int basicPixel = pic.getBasicPixel(col,row);

        //Set the alpha value for the pixel.
        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | opacityMask;

        //Set the modified pixel into tempPicture.
        tempPicture.setBasicPixel(col,row,basicPixel);
      }//end inner loop
    }//end outer loop
```

```
    return tempPicture;

  }//end setOpacity
  //-------------------------------------------------
--//

  //Draws the source picture onto the destination
  // picture with an offset on both axes.
  private void drawPictureOnPicture(
                     Picture source,Picture dest,int
xOff,
                                                  int
yOff){

      Graphics destGraphics = dest.getGraphics();
      Image sourceImage = source.getImage();
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end drawPictureOnPicture method
}//end class Prob07Runner
```

-end-

Java3116 Java OOP Controlling an Edge Detector with a Slider
Learn how to use a slider to continuously change the threshold detection level of an edge detector and to draw the edge-detected image on the screen.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Contents of the file named Prob08.jpg.
- Figure 2 . Output images.

Listings

- Listing 1 . Beginning of the class named Prob08Runner.
- Listing 2 . Beginning of the constructor.
- Listing 3 . Beginning of the edgeDetector method.
- Listing 4 . Detect using adjacent pixels in each column.
- Listing 5 . Register a ChangeEvent listener on the slider.
- Listing 6 . Complete program listing.

Preview

An earlier module *(see 3D Displays, Color Distance, and Edge Detection)* explained how to implement an edge detection algorithm for a fixed detection threshold.

The primary objective of this module is to illustrate how to use a slider to continuously change the detection threshold of an edge detector and to draw the edge-detected image on the screen.

Brief program specifications

Write a program named **Prob08** that uses the class definition for the class named **Prob08** shown in Listing 6 and Ericson's media library along with the image file named Prob08.jpg *(shown in Figure 1)* to produce the two graphic output images shown in Figure 2 .

Figure 1 - Contents of the file named Prob08.jpg.



Figure 2 - Output images.



Note that Figure 2 consists of two separate output images. The image containing the slider is positioned directly below the image of the butterfly.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob08** given in Listing 6 .

The output images

The top image shown in Figure 2 is an image of a butterfly to which an edge detection algorithm has been applied.

The bottom image in Figure 2 is a slider that is used to control the edge-detection threshold.

Detect by rows and by columns

The edge-detection algorithm performs edge detection on a **Picture** object by rows and also by columns.

All edges that are detected by processing adjacent pixels on a row are marked in red. All edges that are detected by processing adjacent pixels on a column are marked in black.

If a pixel is determined to be on an edge using both approaches, it ends up being black. If an edge is not detected, the corresponding pixel is marked in white.

Program behavior

At startup, the thumb on the slider is positioned at the 50-percent mark and the image has been edge-detected using a threshold value of 50.

As you move the slider to the right, the threshold value increases up to a value of 100, which in turn causes the amount of white area in the image to increase.

As you move the slider to the left, the threshold value decreases down to a value of zero, which in turn causes the amount of white area in the image to decrease.

The program must terminate and return control to the operating system when you click the large X in the upper-right corner of the GUI containing the slider.

In addition to the output images described above, your program must display your name on the command-line screen.

Discussion and sample code

The class named Prob08

You can view the driver class named **Prob08** at the beginning of the source code in Listing 6 . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Brief description

Briefly, the **main** method instantiates a new object of the class named **Prob08Runner** and calls the **run** method on that object. When the **run** method returns, the GUI shown in Figure 2 has been displayed on the screen.

At that point, the program simply goes into an idle state and waits for the user to take some action that causes an event to be fired. When an event is fired, it is handled and the program goes idle again waiting for another event.

(Because there are images on the screen, the program does not terminate until the user forces it to terminate.)

The class named Prob08Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 6 near the end of the module.

Beginning of the class named Prob08Runner

The class named **Prob08Runner** begins in Listing 1 .

Listing 1 - Beginning of the class named Prob08Runner.

Listing 1 - Beginning of the class named Prob08Runner.

```
class Prob08Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider();

  private Picture butterfly =
                           new
Picture("Prob08.jpg");

  private int butterflyWidth =

butterfly.getWidth();
  private int butterflyHeight =

butterfly.getHeight();

  private Picture display =
       new
Picture(butterflyWidth,butterflyHeight);

  private Pixel pix1;
  private Pixel pix2;
  private Pixel displayPixel;

  private double distance = 0;
```

There is nothing in Listing 1 that I haven't explained in earlier modules, so I won't repeat those explanations here.

Beginning of the constructor

The constructor begins in Listing 2 .

Listing 2 - Beginning of the constructor.

```
  public Prob08Runner(){//constructor

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                      "Edge Detection
Threshold"));

mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(butterflyWidth + 7,97);
    setLocation(0,butterflyHeight + 25);
    setVisible(true);

    //Produce the initial display with a
threshold
    // value of 50, which matches the initial
    // position of the pointer on the slider.
    display = edgeDetector(butterfly,50);

    display.show();
```

Put the constructor on hold

The only thing that is new in Listing 2 is the call to the method named **edgeDetector** . I will put the explanation of the constructor on hold while I explain the method named **edgeDetector** .

Beginning of the edgeDetector method

The **edgeDetector** method begins in Listing 3 .

This method performs edge detection on a **Picture** object by rows and also by columns.

As mentioned earlier, all edges that are detected by processing adjacent pixels on a row are marked in red.

All edges that are detected by processing adjacent pixels on a column are marked in black.

If a pixel is determined to be on an edge using both approaches, it ends up being black.

Listing 3 - Beginning of the edgeDetector method.

Listing 3 - Beginning of the edgeDetector method.

```
  private Picture edgeDetector(
                        Picture picture,int
threshold){
    for(int row = 0;row < butterflyHeight -
1;row++){
      for(int col = 0;col < butterflyWidth -
1;col++){
        pix1 = picture.getPixel(col,row);
        displayPixel =
display.getPixel(col,row);

        //First process two adjacent pixels on
the
        // same row.
        pix2 = picture.getPixel(col + 1,row);

        //Get and save the color distance
between the
        // two pixels.
        distance =
pix1.colorDistance(pix2.getColor());

        //Compare the color distance to the
threshold
        // and set the color of the pixel in
the
        // display picture accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.RED);
        }else{
          displayPixel.setColor(Color.WHITE);
        }//end else
```

A pair of nested for loops

Listing 3 shows the beginning of a pair of nested **for** loops that I used to detect the edges in the image of the butterfly. The code in Listing 3 detects edges using adjacent pixels on each row and sets the color of the corresponding pixel in the output picture to red if an edge is detected.

*(Note that the output picture referred to by **display** was instantiated as an instance variable in Listing 1 .)*

I explained an edge-detection algorithm in great detail in an earlier module titled 3D Displays, Color Distance, and Edge Detection . The code in Listing 3 is essentially an implementation of that algorithm where the detection threshold is an input parameter to the method.

Detect using adjacent pixels in each column

Listing 4 detects edges using adjacent pixels on the columns.

Listing 4 - Detect using adjacent pixels in each column.

Listing 4 - Detect using adjacent pixels in each column.

```
        //Now process two adjacent pixels in
the same
        // column using the same approach.
        pix2 = picture.getPixel(col,row + 1);
        distance =
pix1.colorDistance(pix2.getColor());
        //Compare the color distance to the
threshold
        // and change pixel color accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.BLACK);
        }//end if

      }//end inner loop
    }//end outer loop

    return display;
  }//end edgeDetector
```

If an edge is detected in Listing 4 , the color of the corresponding pixel in the output picture is set to black.

Return the output picture and terminate

The **edgeDetector** method returns a reference to the output picture and terminates in Listing 4 .

*(While writing this, I realized that because the variable named **display** is an instance variable, the program would also work properly if the **edgeDetector** method were to return void.)*

Returning to the constructor...

Returning to where we left off in Listing 2 , the **show** method is called on the output picture referred to by **display** causing an image similar to that shown in Figure 2 to be displayed.

Register a ChangeEvent listener on the slider

Listing 5 registers an anonymous listener object on the slider. Each time the slider fires a **ChangeEvent** , the method named **stateChanged** is executed.

Listing 5 - Register a ChangeEvent listener on the slider.

```
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent
e){

          display = edgeDetector(

butterfly,slider.getValue());
          display.repaint();

        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-----------------------------------------
------//
  }//end constructor
  //-----------------------------------------
------//
}//end class Prob08Runner
```

Call the edgeDetector method

The **stateChanged** method calls the **edgeDetector** method to get a new edge-detected image for which the threshold is the current value of the slider.

Then the display is repainted showing the new image on the screen.

The end of the program

Listing 5 also signals the end of the constructor, the end of the class, and the end of the program.

Run the program

I encourage you to copy the code from Listing 6 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to use a slider to continuously change the threshold detection level of an edge detector and to draw the edge-detected image on the screen.

What's next?

In the next module, you will learn how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Controlling an Edge Detector with a Slider
- File: Java3116.htm
- Published: 05/13/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is shown in Listing 6 .

Listing 6 - Complete program listing.

```
/*File Prob08 Copyright 2008 R.G.Baldwin
 *Revised 12/31/08

**********************************************
*******/
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.awt.BorderLayout;
import java.awt.Color;

public class Prob08{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    new Prob08Runner();
  }//end main method
}//end class Prob08
//=============================================
======//

class Prob08Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider();

  private Picture butterfly = new
Picture("Prob08.jpg");

  private int butterflyWidth =
butterfly.getWidth();
  private int butterflyHeight =
butterfly.getHeight();
```

```
  private Picture display =
              new
Picture(butterflyWidth,butterflyHeight);

  private Pixel pix1;
  private Pixel pix2;
  private Pixel displayPixel;

  private double distance = 0;

  public Prob08Runner(){//constructor

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                              "Edge Detection
Threshold"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(butterflyWidth + 7,97);
    setLocation(0,butterflyHeight + 25);
    setVisible(true);

    //Produce the initial display with a threshold
value
    // of 50, which matches the initial position
of the
```

```
    // pointer on the slider.
    display = edgeDetector(butterfly,50);

    display.show();
    //-----------------------------------------
------//
    //Register an anonymous listener object on the
slider.
    //Each time the slider fires a ChangeEvent,
this event
    // handler calls the edgeDetector method to
get a new
    // edge-detected image for which the threshold
is the
    // current value of the slider. Then the
display is
    // repainted showing the new image.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          display = edgeDetector(

butterfly,slider.getValue());
          display.repaint();

        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-----------------------------------------
------//
  }//end constructor
  //-----------------------------------------
------//
  /*This method performs edge detection on a
Picture
   *object by rows and also by columns.
```

```
   *All edges that are detected by processing
adjacent
   *pixels on a row are marked in red.
   *All edges that are detected by processing
adjacent
   *pixels on a column are marked in black.
   *If a pixel is determined to be on an edge
using both
   *approaches, it ends up being black.
  */

  private Picture edgeDetector(
                            Picture picture,int
threshold){
    for(int row = 0;row < butterflyHeight -
1;row++){
      for(int col = 0;col < butterflyWidth -
1;col++){
        pix1 = picture.getPixel(col,row);
        displayPixel = display.getPixel(col,row);

        //First process two adjacent pixels on the
same
        // row.
        pix2 = picture.getPixel(col + 1,row);

        //Get and save the color distance between
the two
        // pixels.
        distance =
pix1.colorDistance(pix2.getColor());

        //Compare the color distance to the
threshold and
        // set the color of the pixel in the
display
        // picture accordingly.
```

```
        if(distance > threshold){
          displayPixel.setColor(Color.RED);
        }else{
          displayPixel.setColor(Color.WHITE);
        }//end else

        //Now process two adjacent pixels in the
same
        // column using the same approach.
        pix2 = picture.getPixel(col,row + 1);
        distance =
pix1.colorDistance(pix2.getColor());
        //Compare the color distance to the
threshold and
        // change pixel color accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.BLACK);
        }//end if

      }//end inner loop
    }//end outer loop

    return display;
  }//end edgeDetector
}//end class Prob08Runner
```

-end-

Java3118 Java OOP Controlling an Image-Scaling Program with a Slider
Learn how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- General background information
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Prob09a.jpg.
- Figure 2 . Prob09b.jpg.
- Figure 3 . Output images.

Listings

- Listing 1 . Register a listener object on the slider.
- Listing 2 . The method named drawScaledPictureOnPicture.
- Listing 3 . Complete program listing.

Preview

The primary objective of this module is to illustrate how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

Brief program specifications

Write a program named **Prob09** that uses the class definition for the class named **Prob09** shown in Listing 3 and Ericson's media library along with the image files named Prob09a.jpg *(see Figure 1)* and Prob09b.jpg *(see Figure 2)* to produce the graphic output images shown in Figure 3 .

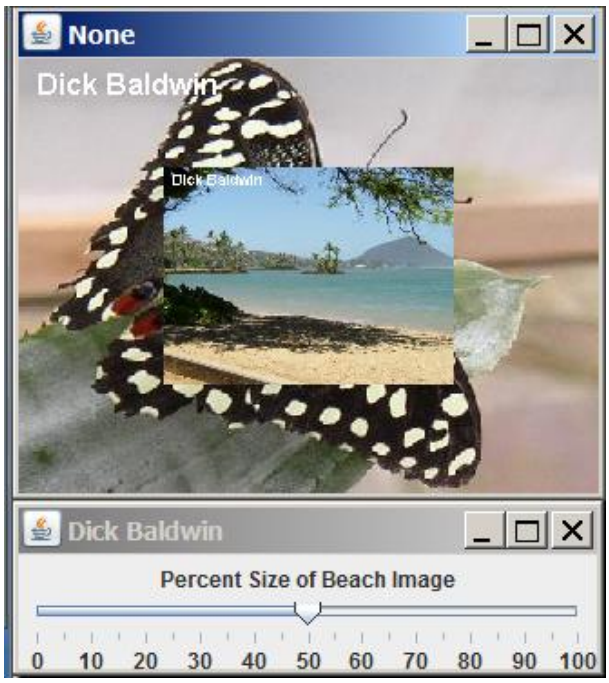
Figure 1 - Prob09a.jpg.



Figure 2 - Prob09b.jpg.



Figure 3 - Output images.



General background information

The overall structure of this program is very similar to a program that I explained in an earlier module titled Controlling an Edge Detector with a Slider . The only significant difference between the two programs is the code that is executed when the slider fires a **ChangeEvent** .

I will explain the code that is new and different in this program and refer you back to the earlier module for an explanation of the remainder of the code. You can view the entire program in Listing 3 near the end of the module.

Discussion and sample code

Register a listener object on the slider

The code in Listing 1 registers an anonymous listener object on the slider.

Listing 1 - Register a listener object on the slider.

```
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          //Restore the background image of the
butterfly.
          graphics = display.getGraphics();
          graphics.drawImage(

butterfly.getImage(),0,0,null);

          drawScaledPictureOnPicture(beach,
                                     display,

slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
```

Each time the slider fires a **ChangeEvent** , the method named **stateChanged** is executed.

Behavior of the stateChanged method

The **stateChanged** method restores the background image of the butterfly. Then it calls the method named **drawScaledPictureOnPicture** to draw a scaled version of the beach on top of the background image using the slider value as the scale factor.

The slider value ranges from 0 to 100. This represents a scale factor as a percent of 1.0. In other words, the beach image is never scaled to a size that is larger than the size of the original beach image.

The image of the beach is always aligned with the center of the **contentPane** of the **JFrame** .

When the **drawScaledPictureOnPicture** method returns, the **repaint** method is called to cause the new image to be rendered on the computer screen.

The method named drawScaledPictureOnPicture

The method named **drawScaledPictureOnPicture** is shown in its entirety in Listing 2 .

Listing 2 - The method named drawScaledPictureOnPicture.

Listing 2 - The method named drawScaledPictureOnPicture.

```
  private void drawScaledPictureOnPicture(
                                      Picture source,
                                      Picture dest,
                                      double
scaleFactor){

    transform = new AffineTransform();

    double translateX = dest.getWidth()/2
                      -
source.getWidth()*scaleFactor/100/2;
    double translateY = dest.getHeight()/2
                     -
source.getHeight()*scaleFactor/100/2;;
    transform.translate(translateX,translateY);

transform.scale(scaleFactor/100.0,scaleFactor/100.0);

    //Get the Graphics2D object used to draw on the
    // destination picture.
    g2 = (Graphics2D)dest.getGraphics();

    //Scale and draw the source image on the
destination
    // image.
    g2.drawImage(source.getImage(),transform,null);

  }//end drawScaledPictureOnPicture method
```

Use an affine transform

This method uses an affine transform to first translate and then scale the source picture and draws the scaled source picture onto the center of the destination picture.

I have published numerous tutorials that explain the use of Affine transforms in Java, including Applying Affine Transforms to Picture Objects . I won't repeat those

explanations in this module, but will simply refer you to the earlier tutorials.

Keywords for a Google search

You can find my other tutorials that explain Affine transforms by using Google to search for the following keywords:

```
richard baldwin java affine transform
```

Run the program

I encourage you to copy the code from Listing 3 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

What's next?

In the next module, you will learn how to use a JSlider object along with Affine Transforms to control the rotation of an image.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Controlling an Image-Scaling Program with a Slider
- File: Java3118.htm
- Published: 05/13/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is shown in Listing 3 below.

Listing 3 - Complete program listing.

```
/*File Prob09 Copyright 2008 R.G.Baldwin
Revised 09/13/10

**************************************************
*/
import java.awt.geom.AffineTransform;
import java.awt.Graphics2D;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
```

```
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class Prob09{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob09Runner();
  }//end main method
}//end class Prob09
//End program specifications.
//////////////////////////////////////////////////////
//
//////////////////////////////////////////////////////
//

/*-----------------------------------------------------
-//

******************************************************
*/
class Prob09Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider(0,100,0);

  private Picture beach = new Picture("Prob09a.jpg");
  private Picture butterfly = new
Picture("Prob09b.jpg");

  private int beachWidth = beach.getWidth();
  private int beachHeight = beach.getHeight();

  private Picture display =
            new Picture(beachWidth,beachHeight);

  private Image image = null;
  private Graphics graphics = null;
  private AffineTransform transform = null;
  private Graphics2D g2 = null;
```

```
  public Prob09Runner(){//constructor

    System.out.println("Dick Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    beach.addMessage("Dick Baldwin",10,20);
    butterfly.addMessage("Dick Baldwin",10,20);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                          "Percent Size of Beach
Image"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);
    //pack();
    setSize(beachWidth + 7,97);
    setTitle("Dick Baldwin");
    setLocation(0,beachHeight + 25);
    setVisible(true);

    //Draw and display the background image of the
    // butterfly.
    graphics = display.getGraphics();
    graphics.drawImage(butterfly.getImage(),0,0,null);

    display.show();
    //-----------------------------------------------
-//
    //Register an anonymous listener object on the
slider.
    //Each time the slider fires a ChangeEvent, this
event
    // handler restores the background image of the
```

```
    // butterfly. Then it draws a scaled version of the
    // beach on top of the background image using the
    // slider value, which ranges from 0 to 100 as the
    // scale factor as a percent of 1.0. The image of
the
    // beach is always aligned with the center
    // of the contentPane of the JFrame.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          //Restore the background image of the
butterfly.
          graphics = display.getGraphics();
          graphics.drawImage(

butterfly.getImage(),0,0,null);

          drawScaledPictureOnPicture(beach,
                                     display,
                                     slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-----------------------------------------------
-//
  }//end constructor
  //-------------------------------------------------
-//

  //Scales and draws the source picture onto the center
  // of the destination picture.
  private void drawScaledPictureOnPicture(
                                      Picture source,
                                      Picture dest,
                                      double
scaleFactor){

    transform = new AffineTransform();
```

```
    double translateX = dest.getWidth()/2
                        -
source.getWidth()*scaleFactor/100/2;
    double translateY = dest.getHeight()/2
                       -
source.getHeight()*scaleFactor/100/2;;
    transform.translate(translateX,translateY);

transform.scale(scaleFactor/100.0,scaleFactor/100.0);

    //Get the Graphics2D object used to draw on the
    // destination picture.
    g2 = (Graphics2D)dest.getGraphics();

    //Scale and draw the source image on the destination
    // image.
    g2.drawImage(source.getImage(),transform,null);

  }//end drawScaledPictureOnPicture method
}//end class Prob09Runner
```

-end-

Java3120 Java OOP Controlling Image Rotation with a Slider and Affine Transforms
Learn to use a JSlider object along with Affine Transforms to control the rotation of an image.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the

figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output for slider at zero.
- Figure 3 . Program output for slider at 240.

Listings

- Listing 1 . Beginning of the class named Prob10Runner.
- Listing 2 . Beginning of the constructor for the Prob10Runner class.
- Listing 3 . Construct the GUI for the slider.
- Listing 4 . Rotate and display the butterfly.
- Listing 5 . Register a ChangeListener on the slider.
- Listing 6 . Beginning of the rotatePicture method.
- Listing 7 . Set up the translation transform.
- Listing 8 . Concatenate the transforms.
- Listing 9 . Transform and draw the butterfly image.
- Listing 10 . Complete program listing.

Preview

In this lecture, I will explain a program that uses **Affine Transforms** to rotate an image by a specified angle around a specified anchor point.

Then the program translates the image so as to center it in a **JFrame** object.

Specification of the rotation angle

A **JSlider** is used to specify the rotation angle.

The range of the slider is from 0 to 360 degrees.

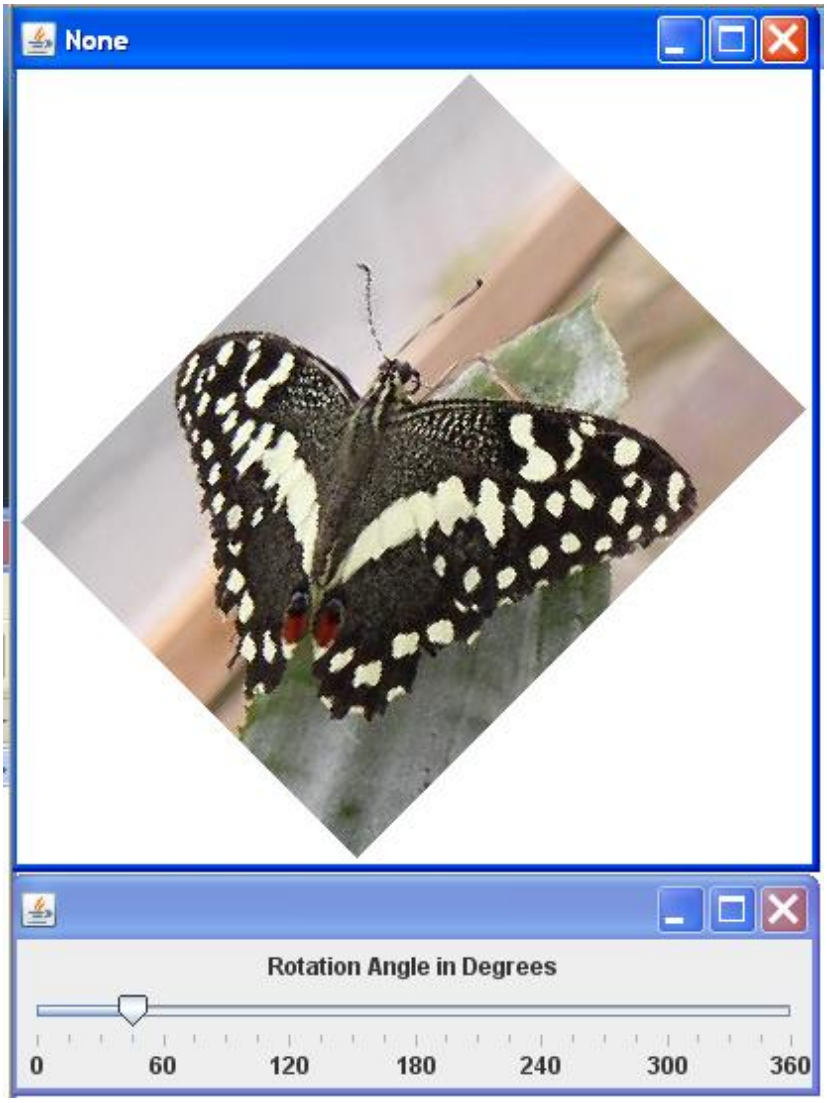
The position of the thumb on the slider specifies a counter-clockwise rotation angle in degrees.

Program output at startup

The thumb on the slider is at 45 degrees when the program starts running.

This causes the initial rotation angle of the butterfly image to be 45 degrees counter-clockwise as shown in Figure 1 .

Figure 1 - Program output at startup.



Corners barely graze inner-edges of the frame's borders

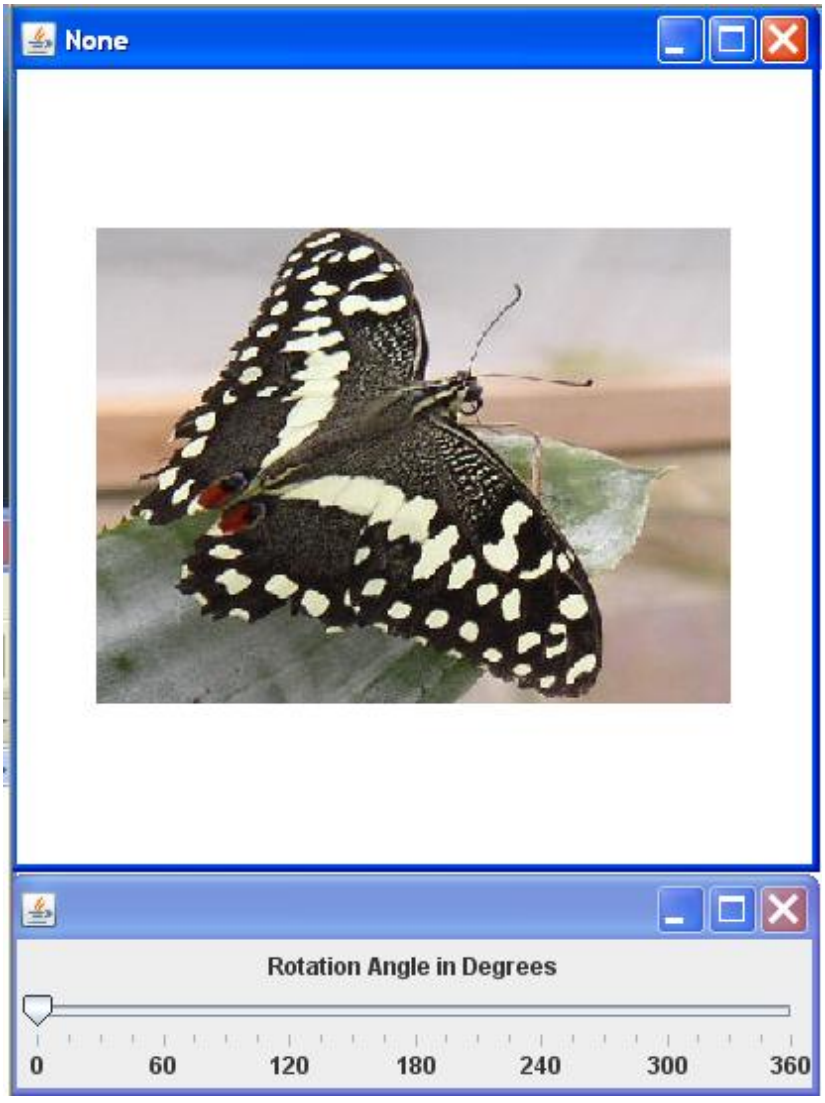
Note that for the initial rotation angle shown in Figure 1 , the corners of the image almost touch the inner-edges of the borders on the frame.

If you run the program and rotate the image, you will see that the size of the frame is such that the corners of the image barely graze the inner-edges for those cases where the diagonals of the image are horizontal and vertical.

Program output for slider at zero degrees

Figure 2 shows the program output when the thumb on the slider has been moved to zero degrees at the far-left end of the slider.

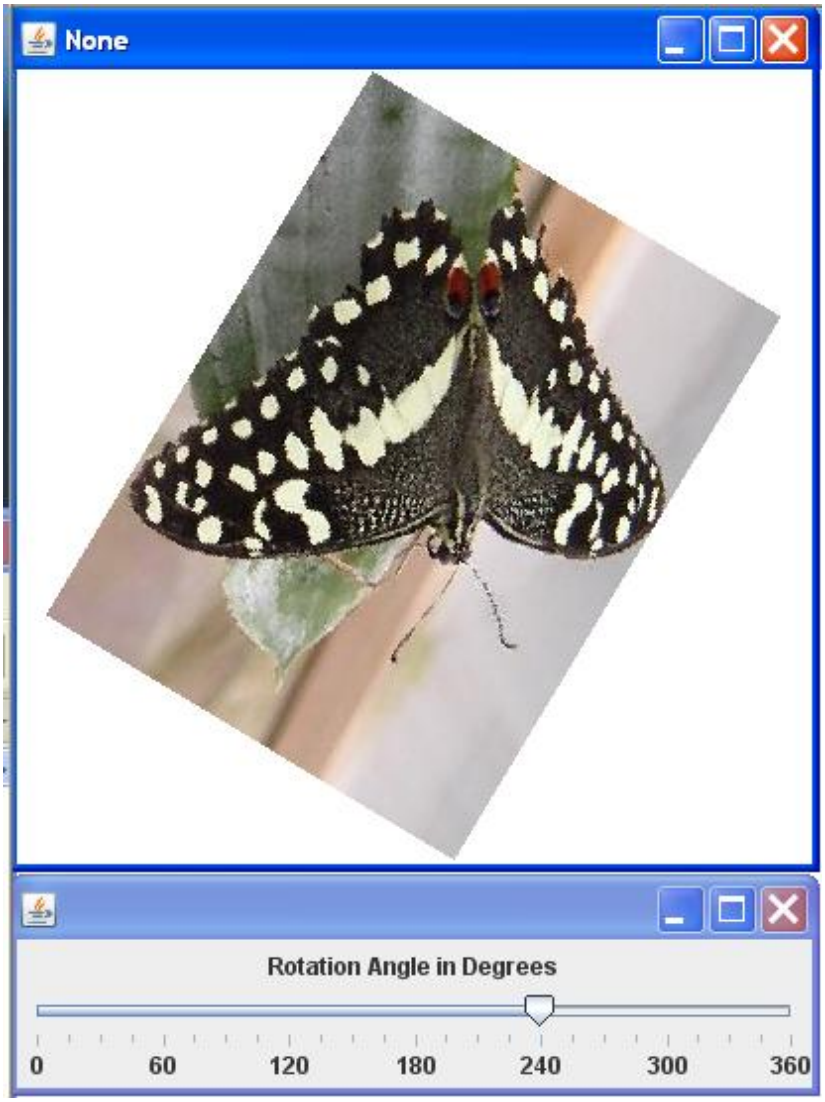
Figure 2 - Program output for slider at zero.



Program output for slider at 240 degrees

Figure 3 shows the result of positioning the thumb on the slider at 240 degrees.

Figure 3 - Program output for slider at 240.



Brief program specifications

Write a program named **Prob10** that uses the **Prob10** class definition shown in Listing 10 and Ericson's media library along with the image file named Prob10.jpg to produce the graphic output images shown in Figure 1 through Figure 3 .

The butterfly image rotates smoothly around its center as you move the slider back and forth.

The program must terminate and return control to the operating system when you click the large X in the upper-right corner of the GUI containing the slider.

General background information

The overall structure of this program is very similar to programs that I explained in earlier lectures titled

- Controlling an Edge Detector with a Slider , and
- Controlling an Image-Scaling Program with a Slider .

The most significant difference...

Is the code that is executed when the slider fires a **ChangeEvent** . There are a few other differences as well.

New and different code

I will explain the code that is new and different in this program.

I will refer you back to the earlier lectures for an explanation of the remainder of the code.

You can view the entire program in Listing 10 .

Discussion and sample code

Beginning of the class named Prob10Runner

The code in the **main** method in Listing 10 instantiates a new object of the class named **Prob10Runner** .

Listing 1 shows the beginning of the class named **Prob10Runner** .

Listing 1 declares several instance variables and initializes some of them.

I will refer back to some of these variables in later paragraphs.

Listing 1 - Beginning of the class named Prob10Runner.

Listing 1 - Beginning of the class named Prob10Runner.

```
class Prob10Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();

  //Instantiate a new slider setting the limits
and the
  // initial position of the thumb.
  private JSlider slider = new
JSlider(0,360,45);

  private Picture butterfly = new
Picture("Prob10.jpg");
  private Picture background = null;

  private int butterflyWidth =
butterfly.getWidth();
  private int butterflyHeight =
butterfly.getHeight();

  private Picture display = null;
  private int displayWidth = 0;
  private int displayHeight = 0;

  private Image image = null;
  private Graphics graphics = null;
  private AffineTransform rotateTransform =
null;
  private AffineTransform translateTransform =
null;
  private Graphics2D g2 = null;
```

Beginning of the constructor for the Prob10Runner class

Listing 2 shows the beginning of the constructor for the class named **Prob10Runner** .

Listing 2 - Beginning of the constructor for the Prob10Runner class.

```
  public Prob10Runner(){//constructor


setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Compute the minimum dimensions allowed
for the
    // display window that will contain the
butterfly
    // image rotated at any angle. This turns
out to
    // be a square with the length of each side
equal
    // to the diagonal length of the butterfly
picture.
    //The length of each side was increased by
one
    // pixel to guard against loss of precision
when
    // converting from double to int.
    int diagonal = 1 + (int)(Math.sqrt(

butterflyWidth*butterflyWidth +

butterflyHeight*butterflyHeight));
```

Listing 2 - Beginning of the constructor for the Prob10Runner class.

```
    //Instantiate the picture in which the
rotated
    // butterfly image will be displayed.
    display = new Picture(diagonal,diagonal);
    displayWidth = displayHeight = diagonal;

    //This picture provides a white background
the same
    // size as the display picture.
    background = new
Picture(diagonal,diagonal);
```

Not much that is new here

The only thing that might be considered new in Listing 2 is the code that computes the minimum dimensions for a display window that will contain the butterfly image rotated by any angle.

The rationale for this computation is explained in the comments.

Construct the GUI for the slider

The code in Listing 3 constructs the GUI that contains the slider shown in Figure 1 .

Although the code in Listing 3 is a little tedious, there are no new concepts associated with that code.

Listing 3 - Construct the GUI for the slider.

Listing 3 - Construct the GUI for the slider.

```
    //Construct the GUI for the slider.
    slider.setMajorTickSpacing(60);
    slider.setMinorTickSpacing(15);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                              "Rotation Angle in
Degrees"));

mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);
    pack();//Adjust the size of the slider GUI.

    //Compute an improved size and location for
the
    // GUI containing the slider.
    setSize(displayWidth + 2 *
getInsets().left,
            mainPanel.getHeight() +
slider.getHeight());
    setLocation(0,displayHeight +
getInsets().top
                                  +
getInsets().bottom + 1);
    setVisible(true);
```

Rotate and display the butterfly

The code in Listing 4 calls the method named **rotatePicture** , passing the initial value of the thumb on the slider as a parameter to cause the initial display of the butterfly to be properly rotated.

Then Listing 4 calls the **show** method on the display to cause the rotated butterfly image to be displayed as shown in Figure 1 .

Listing 4 - Rotate and display the butterfly.

```
rotatePicture(slider.getValue());

display.show();
```

Register a ChangeListener on the slider

The code in Listing 5 registers an anonymous **ChangeListener** object on the slider.

Listing 5 - Register a ChangeListener on the slider.

Listing 5 - Register a ChangeListener on the slider.

```
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e)
{
          //Restore the background image of the
display
          // to all white.
          graphics = display.getGraphics();
          graphics.drawImage(

background.getImage(),0,0,null);

          //Rotate the butterfly image, draw it
on the
          // display, and repaint the display
on the
          // screen..
          rotatePicture(slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-----------------------------------------
---------//
  }//end constructor
```

Each time the slider fires a **ChangeEvent** , the **stateChanged** method in Listing 5 is executed.

The **stateChanged** method begins by restoring the background image of thedisplay.

Then the **stateChanged** method calls the **rotatePicture** method to draw a rotated version of thebutterfly on top of the background image using theslider value, *(which ranges from 0 to +360)* , asthe rotation angle in degrees.

The image of thebutterfly *(see Figure 1*) is always centered in the displaypicture.

Finally, the **stateChanged** method causes the display to be repainted to force the rotated image to appear on the screen.

End of the constructor

Listing 5 signals the end of the constructor for the class named **Prob10Runner** .

Beginning of the rotatePicture method

The **rotatePicture** method begins in Listing 6 .

Listing 6 - Beginning of the rotatePicture method.

Listing 6 - Beginning of the rotatePicture method.

```
  private void rotatePicture(double angle){

    //Set up the rotation transform
    rotateTransform = new AffineTransform();
    //Negate the angle for counter-clockwise
rotation.
    rotateTransform.rotate(-
Math.toRadians(angle),
                           butterflyWidth/2,
                           butterflyHeight/2);
```

The **rotatePicture** method accepts a rotation angle in degrees as an incoming parameter.

The **rotatePicture** method

- rotates a butterfly image by that angle around its center,
- translates the rotated image, and
- draws the rotated image in the center of a display picture.

Set up the rotation transform

Listing 6 begins by instantiating a new object of the **AffineTransform** class.

Then it calls the **rotate** method on the transform object to configure that object for use in rotating the image of the butterfly.

Overloaded versions of the *rotate* method

There are several overloaded versions of the **rotate** method.

The version called in Listing 6 requires three parameters:

- The rotation angle in radians

- The X coordinate of the anchor point *(point around which the rotation will take place)* .
- The Y coordinate of the anchor point.

A positive rotation value indicates a clockwise rotation.

Convert from degrees to radians

Listing 6 converts the incoming angle in degrees to radians

The code in Listing 6 applies a minus sign to convert the angle from a positive value to a negative value for ***counter-clockwise*** rotation, and passes that value as the rotation parameter.

Specify the rotation *anchor point*

Listing 6 also specifies the center of the butterfly image as the anchor point.

Set up the translation transform

Listing 7 sets up the translation transform that will be used to translate the rotated image to the center of the new **Picture** object.

Listing 7 - Set up the translation transform.

```
    translateTransform = new AffineTransform();
    translateTransform.translate(
                        (displayWidth -
butterflyWidth)/2,
                        (displayHeight -
butterflyHeight)/2);
```

Listing 7 instantiates a new **AffineTransform** object.

Then it calls the **translate** method to configure that object for use in translating *(moving)* the rotated image of the butterfly.

The translation transform object will be concatenated with the rotation object from Listing 6 to produce an **AffineTransform** object that can ***rotate*** and ***translate*** , in that order.

Concatenate the transforms

Listing 8 concatenates the **AffineTransform** object referred to by **translateTransform** with the transform object referred to by **rotateTransform** .

Listing 8 - Concatenate the transforms.

```
translateTransform.concatenate(rotateTransform);
```

The resulting composite transform

The concatenation in Listing 8 results in an **Affine Transform** object referred to by **translateTransform** that will first rotate the image around its center and then translate the rotated image to the center of the new **Picture** object.

The order of operations is very important

When rotating and translating images, it is important that the two operations be performed in the correct order.

Otherwise, the results might not be what you want.

Transform and draw the butterfly

Listing 9 applies the composite transform to the butterfly image and draws the transformed image on the output picture as shown in Figure 1 , Figure 2 , and Figure 3 .

Listing 9 - Transform and draw the butterfly image.

```
    Graphics2D g2 =
(Graphics2D)display.getGraphics();
    g2.drawImage(butterfly.getImage(),
                 translateTransform,
                 null);

  }//end rotatePicture

}//end class Prob10Runner
```

The end of the program

Listing 9 also signals the end of the **rotatePicture** method, the end of the **Prob10Runner** class, and the end of the program.

Run the program

I encourage you to copy the code from Listing 10 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this lecture, you learned how to use a **JSlider** object along with **Affine Transforms** to control the rotation of an image.

What's next?

In the next module, you will learn how to open an image file *(specified by a string in a text field)* in a **PictureExplorer** object.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Controlling Image Rotation with a Slider and Affine Transforms
- File: Java3120.htm
- Published: 09/07/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com

showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this lecture is shown in Listing 10 below.

Listing 10 - Complete program listing.

```
/*File Prob10 Copyright 2008 R.G.Baldwin
Revised 09/14/10

**********************************************************
******/
import java.awt.geom.AffineTransform;
import java.awt.Graphics2D;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class Prob10{
  public static void main(String[] args){
```

```
    new Prob10Runner();
  }//end main method
}//end class Prob10
/*************************************************
******/

class Prob10Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();

  //Instantiate a new slider setting the limits and
the
  // initial position of the thumb.
  private JSlider slider = new JSlider(0,360,45);

  private Picture butterfly = new
Picture("Prob10.jpg");
  private Picture background = null;

  private int butterflyWidth =
butterfly.getWidth();
  private int butterflyHeight =
butterfly.getHeight();

  private Picture display = null;
  private int displayWidth = 0;
  private int displayHeight = 0;

  private Image image = null;
  private Graphics graphics = null;
  private AffineTransform rotateTransform = null;
  private AffineTransform translateTransform =
null;
  private Graphics2D g2 = null;

  public Prob10Runner(){//constructor
```

```
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Compute the minimum dimensions allowed for
the
    // display window that will contain the
butterfly
    // image rotated at any angle. This turns out
to
    // be a square with the length of each side
equal
    // to the diagonal length of the butterfly
picture.
    //The length of each side was increased by one
    // pixel to guard against loss of precision
when
    // converting from double to int.
    int diagonal = 1 + (int)(Math.sqrt(

butterflyWidth*butterflyWidth +

butterflyHeight*butterflyHeight));

    //Instantiate the picture in which the rotated
    // butterfly image will be displayed.
    display = new Picture(diagonal,diagonal);
    displayWidth = displayHeight = diagonal;

    //This picture provides a white background the
same
    // size as the display picture.
    background = new Picture(diagonal,diagonal);

    //Construct the GUI for the slider.
    slider.setMajorTickSpacing(60);
    slider.setMinorTickSpacing(15);
    slider.setPaintTicks(true);
```

```
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                            "Rotation Angle in
Degrees"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);
    pack();//Adjust the size of the slider GUI.

    //Compute an improved size and location for the
    // GUI containing the slider.
    setSize(displayWidth + 2 * getInsets().left,
            mainPanel.getHeight() +
slider.getHeight());
    setLocation(0,displayHeight + getInsets().top
                                +
getInsets().bottom + 1);
    setVisible(true);

    //Place the butterfly image in the display
picture
    // with a rotation angle specified by the
initial
    // position of the thumb on the slider.

    rotatePicture(slider.getValue());

    display.show();
    //-----------------------------------------
-----//
    //Register an anonymous listener object on the
slider.
    //Each time the slider fires a ChangeEvent,
this event
```

```
    // handler restores the background image of the
    // display. Then it draws a rotated version of
the
    // butterfly on top of the background image
using the
    // slider value, which ranges from 0 to +360,
as
    // the rotation angle in degrees. The image of
the
    // butterfly is always centered in the display
    // picture.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          //Restore the background image of the
display
          // to all white.
          graphics = display.getGraphics();
          graphics.drawImage(

background.getImage(),0,0,null);
          //Rotate the butterfly image, draw it on
the
          // display, and repaint the display on
the
          // screen..
          rotatePicture(slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------
-----//
  }//end constructor
  //-----------------------------------------------
-----//
```

```
  //This method accepts a rotation angle in
degrees. It
  // rotates a butterfly image by that angle around
its
  // center, translates, and draws the rotated
image in
  // the center of a display picture.
  private void rotatePicture(double angle){

    //Set up the rotation transform
    rotateTransform = new AffineTransform();
    //Negate the angle for counter-clockwise
rotation.
    rotateTransform.rotate(-Math.toRadians(angle),
                           butterflyWidth/2,
                           butterflyHeight/2);

    //Set up the translation transform that will
translate
    // the rotated image to the center of the new
Picture
    // object.
    translateTransform = new AffineTransform();
    translateTransform.translate(
                      (displayWidth -
butterflyWidth)/2,
                      (displayHeight -
butterflyHeight)/2);

    //Concatenate the two transforms so that the
image
    // will first be rotated around its center and
then
    // translated to the center of the new Picture
object.

translateTransform.concatenate(rotateTransform);
```

```
    //Get the graphics context of the display
picture,
    // apply the transform to the butterfly image,
and
    // draw the transformed picture on the display
    // picture.
    Graphics2D g2 =
(Graphics2D)display.getGraphics();
    g2.drawImage(butterfly.getImage(),
                 translateTransform,
                 null);

  }//end rotatePicture

}//end class Prob10Runner
```

-end-

Java3122 Java OOP Opening an Image File in a PictureExplorer Object
Learn how to open an image file specified by a string in a text field in a PictureExplorer object.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Complete program listing
- Will explain the code in fragments
- Run the program
- What's next?
- Miscellaneous

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Screen shot after the image is loaded.

Listings

- Listing 1 . Complete program listing.
- Listing 2 . The driver class.
- Listing 3 . Beginning of the Prob11Runner class.
- Listing 4 . Beginning of the constructor.
- Listing 5 . Beginning of anonymous listener class.
- Listing 6 . Completion of the anonymous listener class.

Preview

This program demonstrates how to specify an image file in a text field, and open the image in a new **PictureExplorer** object.

Program output at startup

Figure 1 shows the program output at startup.

Figure 1 - Program output at startup.



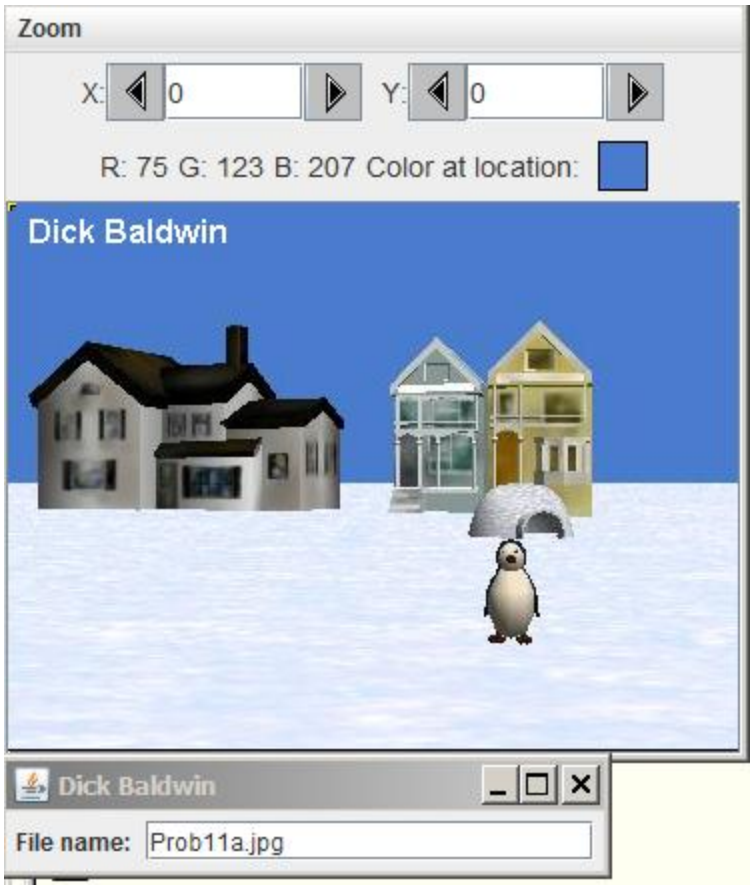
Pre-loaded image file name

For convenience, the text field is pre-loaded with the name of an image file that is located in the current directory.

Press Enter to load the image

Pressing the Enter key while the text field has the focus will cause the image to be loaded into a **PictureExplorer** object, and will cause the **PictureExplorer** object to be displayed as shown in Figure 2 .

Figure 2 - Screen shot after the image is loaded.



Control panel is relocated

Note in Figure 2 that the control panel containing the text field has been automatically relocated to a position immediately below the **PictureExplorer** object.

Another image file in the current directory

Entering the path and name of any jpeg file will cause that file to be loaded into a new **PictureExplorer** object.

Complete program listing

A complete listing of the program discussed in this lecture is shown in Listing 1 below.

Listing 1 - Complete program listing.

```
/*File Prob11 Copyright 2012 R.G.Baldwin
The purpose of this program is to demonstrate the
use of
a JTextField object to specify the name of an
image file,
which is then loaded and displayed in a
PictureExplorer
object.

The text field is pre-loaded with the name of an
image
file that is in the current directory
(Prob11a.jpg).

An image file named Prob11b.jpg is also in the
current
directory and can be loaded and displayed by
entering
the name in the text field.

Any image file on the disk can be loaded and
displayed
by entering the path and name of the image file.
**********************************************
*******/

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.WindowConstants;

public class Prob11{
  public static void main(String[] args){
    new Prob11Runner();
  }//end main method
}//end class Prob11
//===========================================
======//

class Prob11Runner extends JFrame{
  private Prob11Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel fileNamePanel = new JPanel();
  private JTextField inputFileNameField =
                           new
JTextField("Prob11a.jpg",20);

  private String fileName = "no file specified";
  //-----------------------------------------
------//

  public Prob11Runner(){//constructor
    fileNamePanel.add(new JLabel("File name: "));
    fileNamePanel.add(inputFileNameField);

    //Add the fileNamePanel to the content pane,
adjust
    // to the correct size, and set the title.
    getContentPane().add(fileNamePanel);
    pack();
    setTitle("Dick Baldwin");
```

```
    //Make the GUI visible, set the focus, and
establish
    // a reference to the GUI object.
    setVisible(true);
    inputFileNameField.requestFocus();
    jFrameObj = this;

    //-----------------------------------------
------//
    //Register listeners on the user input field.
    //-----------------------------------------
------//

    inputFileNameField.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          fileName = inputFileNameField.getText();

          pix = new Picture(fileName);
          pix.addMessage("Dick Baldwin",10,20);
          explorer = new PictureExplorer(pix);

          //Set the location for the control GUI
          // immediately below the PictureExplorer
object,
          // and set its default close operation.
          setLocation(0,pix.getHeight() + 128);
          jFrameObj.setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
------//
  }//end constructor
```

```
  //-----------------------------------------------
------//
}//end class Prob11Runner
```

Will explain the code in fragments

As usual, I will break the code down, and explain it in fragments.

The driver class

The driver class is shown in Listing 2 .

There is nothing new in Listing 2 .

Listing 2 - The driver class.

```
public class Prob11{
  public static void main(String[] args){
    new Prob11Runner();
  }//end main method
}//end class Prob11
```

Beginning of the Prob11Runner class

The **Prob11Runner** class begins in Listing 3 .

Listing 3 declares several instance variables that will be used later in the program, initializing some of them.

Note that the class extends **JFrame** , so an object of the class *"is a"* **JFrame** object.

Listing 3 - Beginning of the Prob11Runner class.

```
class Prob11Runner extends JFrame{
  private Prob11Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel fileNamePanel = new JPanel();
  private JTextField inputFileNameField =
                new
JTextField("Prob11a.jpg",20);

  private String fileName = "no file
specified";
```

Beginning of the constructor

The constructor for the **Prob11Runner** class begins in Listing 4 .

The code in Listing 4 performs the physical construction of the GUI show in Figure 1 .

The code in Listing 4 saves a reference to the object so that it can be accessed later from within a listener object.

Listing 4 - Beginning of the constructor.

Listing 4 - Beginning of the constructor.

```
  public Prob11Runner(){//constructor
    fileNamePanel.add(new JLabel("File name:
"));
    fileNamePanel.add(inputFileNameField);

    //Add the fileNamePanel to the content
pane, adjust
    // to the correct size, and set the title.
    getContentPane().add(fileNamePanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible, set the focus, and
establish
    // a reference to the GUI object.
    setVisible(true);
    inputFileNameField.requestFocus();
    jFrameObj = this;
```

Register a listener on the text field

Listing 5 shows the beginning of code that instantiates an **ActionListener** object of an anonymous listener class, and registers the listener object on the text field shown in Figure 1 .

Listing 5 - Beginning of anonymous listener class.

Listing 5 - Beginning of anonymous listener class.

```
    inputFileNameField.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          fileName =
inputFileNameField.getText();

          pix = new Picture(fileName);
          pix.addMessage("Dick
Baldwin",10,20);

          explorer = new PictureExplorer(pix);
```

Get image and create Picture object

Listing 5 begins by getting the name of the image file from the text field, using that image file to create a new **Picture** object, and adding a name in the upper-left corner of the picture.

Create new PictureExplorer object

Then Listing 5 instantiates a new **PictureExplorer** object that encapsulates the **Picture** object that was created from the image file.

Completion of the anonymous listener class

Listing 6 shows the completion of the anonymous listener class, as well as the end of the constructor, and the end of the class named **Prob11Runner** .

Listing 6 - Completion of the anonymous listener class.

```
          setLocation(0,pix.getHeight() +
128);

          jFrameObj.setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
----------//
  }//end constructor
  //-----------------------------------------
----------//
}//end class Prob11Runner
```

Relocate the GUI

Listing 6 begins by setting the location of the GUI to a location that is immediately below the **PictureExplorer** object.

Relocate the GUI (cont'd)

Note that the value of 128 pixels was experimentally determined to be the approximate difference between the height of the **PictureExplorer** object and the height of the **Picture** object that it encapsulates.

(This difference may not be correct for different display options on different operating systems.)

Set the default close operation

Listing 6 ends by setting the behavior of the X-button in the GUI to a value that will cause it to terminate the program when the button is clicked.

Run the program

I encourage you to copy the code from Listing 1 and download the image file named Prob11a.jpg . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

What's next?

In the next module, you will learn how to extract color data from a selected pixel in a **PictureExplorer** object, and how to display a color value in a text field. You will also learn how to disable the X-button in the **PictureExplorer** object, and how to use a reference to the **JFrame** object that serves as a container for the **PictureExplorer** object.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Opening an Image File in a PictureExplorer Object
- File: Java3122.htm
- Published: 09/08/12
- Revised: 02/06/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java3124 Java OOP Extracting pixel color data from a PictureExplorer object
Learn how to extract color data from a selected pixel in a PictureExplorer object, and to display the value in a text field. Also learn how to disable the X-button in the PictureExplorer object, and how to use a reference to the JFrame object that serves as a container for the PictureExplorer object.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listings

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output after clicking the button.

Listings

- Listing 1 . Addition of getter methods.
- Listing 2 . Disabled the call to setDefaultCloseOperation method.
- Listing 3 . The driver class for Prob12.
- Listing 4 . Beginning of the class named Prob12Runner.
- Listing 5 . Beginning of the constructor.
- Listing 6 . Construct a Picture object.
- Listing 7 . Construct a PictureExplorer object.
- Listing 8 . Set the size and location of the GUI control panel.
- Listing 9 . Register a listener object on the button.
- Listing 10 . Prob12.java.
- Listing 11 . Modified PictureExplorer.java.

Preview

In this module, I will show you how to extract color data from a selected pixel in a **PictureExplorer** object, and how to display the value in a text field.

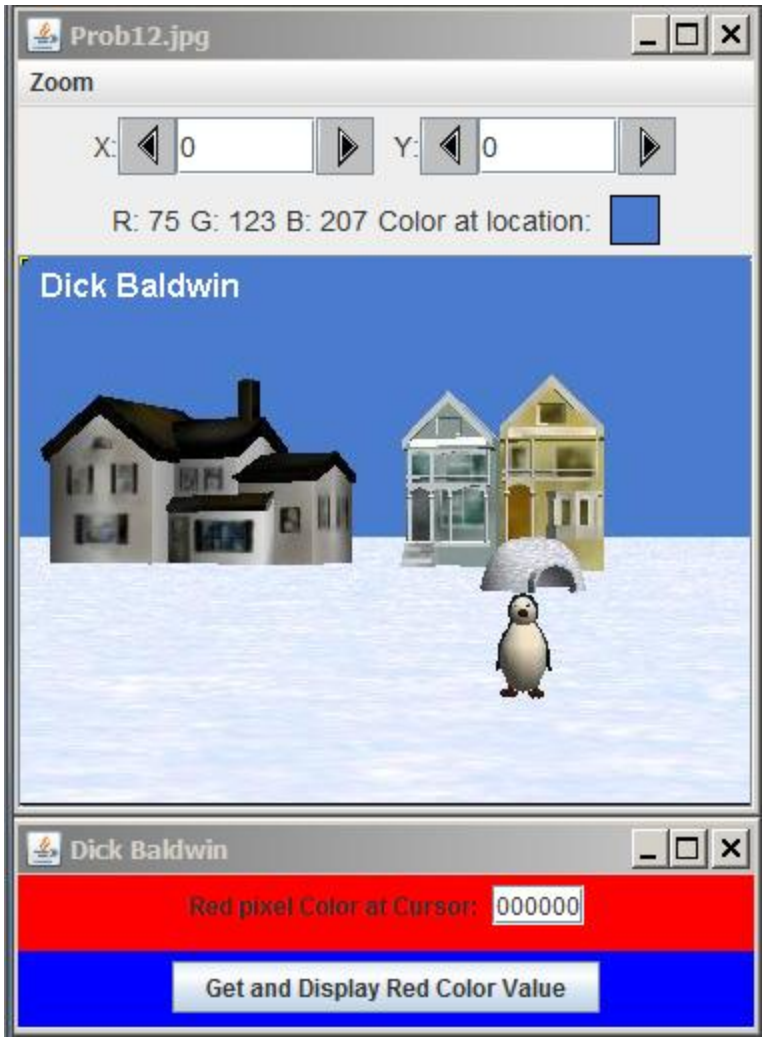
I will also show you how to disable the **X-button** in the **PictureExplorer** object, and how to get and use a reference to the **JFrame** object that serves as a container for the **PictureExplorer** object.

Program output at startup

Figure 1 shows a screen shot of the program output at startup.

Note that the text field in the red panel near the bottom contains all zeros.

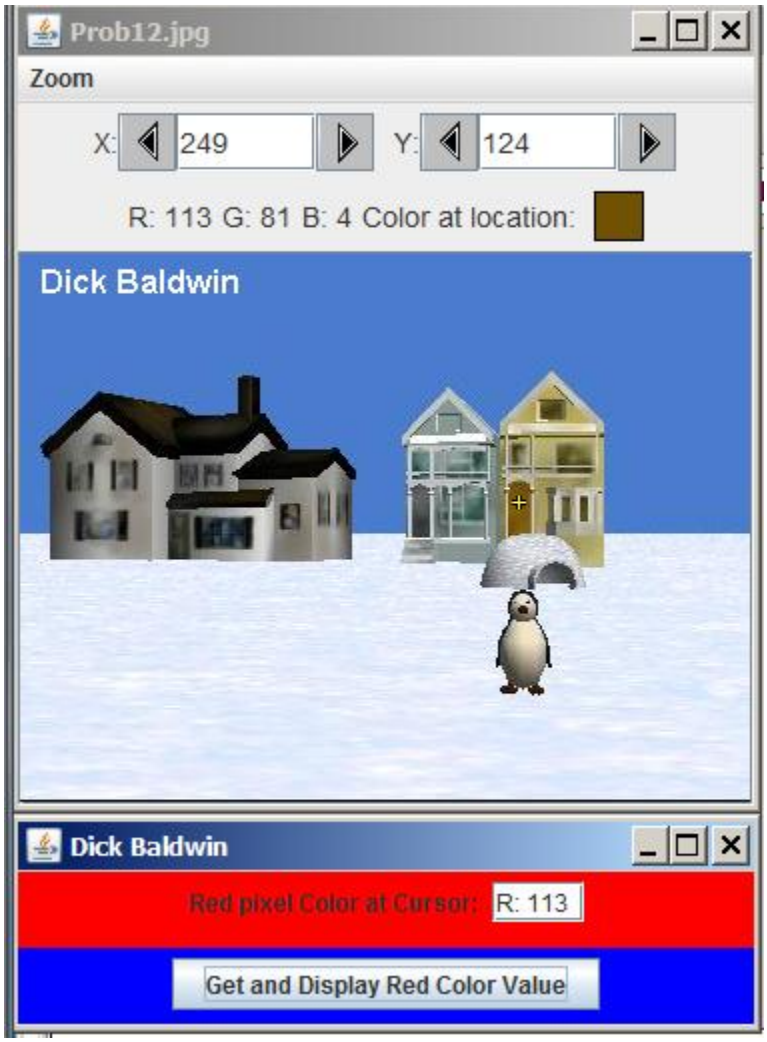
Figure 1 - Program output at startup.



Program output after clicking the button

Figure 2 shows the screen output after moving the **PictureExplorer** crosshair cursor to the brown door immediately above the penguin, and clicking the button labeled *"Get and Display Red Color Value."*

Figure 2 - Program output after clicking the button.



The red color value

The text field near the bottom of Figure 2 displays the red color value of the pixel at the location of the crosshair cursor on the door in the image.

Compare this value with the value of the red color value displayed near the top of the **PictureExplorer** object. You will see that they match.

Discussion and sample code

Two source code files are required for this program:

- The file named **Prob12.java**
- A modified version of Ericson's file named **PictureExplorer.java**

A complete listing of **Prob12.java** is shown in Listing 10 .

A complete listing of the modified **PictureExplorer.java** file is shown in Listing 11 .

Modifications to PictureExplorer.java

The modifications that I made can be found by searching the source code in Listing 11 for the word "Baldwin".

The modifications were:

- I added a *getter* method to cause the red color value to be accessible from outside the **PictureExplorer** object.
- I added a *getter* method that returns a reference to the **JFrame** object containing the **PictureExplorer** object.
- I disabled the call to the **setDefaultCloseOperation** method for the **PictureExplorer** object.

Addition of getter methods

The code for the each of the new *getter* methods is shown in Listing 1 .

These two methods simply return values that already exist in the **PictureExplorer** object.

Listing 1 - Addition of getter methods.

Listing 1 - Addition of getter methods.

```
//===Methods added by Baldwin on
05/15/12==============//
  //Method to get the red color value as text.
  public String getRValue(){
    return rValue.getText();
  }//end getRValue

  //Method to get a reference to the JFrame
containing
  // the PictureExplorer object.
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()
//===End methods added by Baldwin on
05/15/12===========//
```

Disabled the call to setDefaultCloseOperation method

Comment indicators were used to disable the call to the **setDefaultCloseOperation** method as shown in Listing 2 .

This code was replaced by code that you will see later that causes the **JFrame** to *do nothing* when the X-button on the **PictureExplorer** object is clicked.

Listing 2 - Disabled the call to setDefaultCloseOperation method.

Listing 2 - Disabled the call to setDefaultCloseOperation method.

```
/*Disabled by Baldwin on 5/15/12
   pictureFrame.setDefaultCloseOperation(

JFrame.DISPOSE_ON_CLOSE);
*/
```

No more modifications

That concludes the discussion of modifications to Ericson's **PictureExplorer** source code.

The driver class named Prob12

The driver class is shown in Listing 3 .

There is nothing new here.

Listing 3 - The driver class for Prob12.

```
public class Prob12{
  public static void main(String[] args){
    new Prob12Runner();
  }//end main method
}//end class Prob12
```

Beginning of the class named Prob12Runner

The class named **Prob12Runner** begins in Listing 4 .

The code in Listing 4 declares several instance variables and initializes some of them.

Listing 4 - Beginning of the class named Prob12Runner.

```
class Prob12Runner extends JFrame{
  private JFrame explorerFrame = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();

  private JTextField redField = new
JTextField("000000");

  private JButton getDataButton = new JButton(
                        "Get and Display Red
Color Value");
  private String fileName = "Prob12.jpg";
```

Beginning of the constructor

The constructor for the class named **Prob12Runner** begins in Listing 5 .

The code in Listing 5 performs the physical construction of the GUI control panel at the bottom of Figure 1 .

The final position of the GUI control panel will be established after the size of the **PictureExplorer** object's **JFrame** is known.

Listing 5 - Beginning of the constructor.

```
    controlPanel.setLayout(new
GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.RED);
    colorPanel.add(new JLabel(
                          "Red pixel Color at
Cursor: "));
    colorPanel.add(redField);

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(getDataButton);

    getContentPane().add(controlPanel);
    setTitle("Dick Baldwin");

    setVisible(true);
```

Construct a Picture object

Listing 6 creates a new **Picture** object, and adds a name to the picture before using it to construct a **PictureExplorer** object.

Listing 6 - Construct a Picture object.

```
    pix = new Picture(fileName);
    pix.addMessage("Dick Baldwin",10,20);
```

Construct a PictureExplorer object

Listing 7 begins by constructing a new **PictureExplorer** object that encapsulates the **Picture** object instantiated in Listing 6 , and by saving a reference to the **PictureExplorer** object in the instance variable named **explorer** .

Listing 7 - Construct a PictureExplorer object.

```
    explorer = new PictureExplorer(pix);

    explorerFrame = explorer.getFrame();

    explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);
```

Get and use a reference to the JFrame object

Then Listing 7 calls the **PictureExplorer** object's new **getFrame** method to get a reference to the **JFrame** object that contains the **PictureExplorer** .

This reference is used in Listing 7 to call the **setDefaultCloseOperation** method to cause the **JFrame** to *do nothing* when the X-button on the **JFrame** is clicked.

The reference is also used in Listing 8 to establish the size and location of the GUI control panel.

Set the size and location of the GUI control panel

Now that the **PictureExplorer** object exists, Listing 8 sets the size and location of the GUI control panel as shown in Figure 1 .

Listing 8 - Set the size and location of the GUI control panel.

```
    //Set the size of the control GUI.
    setSize(explorerFrame.getWidth(),110);

    //Set the location for the control GUI
    setLocation(0,explorerFrame.getHeight());

    //Set the default close operation for the
control GUI.
    setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);
```

Note that it isn't necessary to guess about the actual height of the **JFrame** as was the case in an earlier module. The reference to the **JFrame** object provides that information.

Set the default close operation for the control GUI

Finally, Listing 8 sets the default close operation on the control GUI to cause the program to terminate when the user clicks the X-button on the GUI.

Register a listener object on the button

Listing 9 registers an **ActionListener** object on the button to cause it to call the **PictureExplorer** object's new **getRValue** method to get the red color value for the pixel at the current **PictureExplorer** crosshair cursor position, and to display that value in the text field as shown in Figure 2 when the user clicks the button.

Listing 9 - Register a listener object on the button.

```
    getDataButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){


redField.setText(explorer.getRValue());

        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
----------//

  }//end constructor
}//end class Prob12Runner
```

End of the program

Listing 9 also signals the end of the constructor and the end of the class named **Prob12Runner** .

Run the program

I encourage you to copy the code from Listing 10 and Listing 11 and download the image file named Prob12.jpg . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

You learned how to extract color data from a selected pixel in a **PictureExplorer** object, and to display the value in a text field.

You also learned how to disable the X-button in the **PictureExplorer** object, and how to use a reference to the **JFrame** object that serves as a container for the PictureExplorer object.

What's next?

In the next module, you will learn how to handle document events on text fields containing color values. You will also learn how to create a color swatch.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Extracting pixel color data from a PictureExplorer object
- File: Java3124.htm
- Published: 09/08/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the source code discussed in this module are shown in Listing 10 and Listing 11 below.

Listing 10 - Prob12.java.

```
/*File Prob12 Copyright 2012 R.G.Baldwin
The purpose of this program is to demonstrate how
```

```
to get
a color value from a PictureExplorer object and to
display
it in JTextField object.

It also demonstrates how to access and use a
reference
to a JFrame object that serves as the container
for a
PictureExplorer object.

Finally, it demonstrates how to disable the X-
button in
a PictureExplorer object.

Note that this program requires access to a
modified
version of the PictureExplorier class.
**************************************************
*******/

import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.Color;


public class Prob12{
  public static void main(String[] args){
    new Prob12Runner();
  }//end main method
```

```
}//end class Prob12
//===================================================
======//

class Prob12Runner extends JFrame{
  private JFrame explorerFrame = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();

  private JTextField redField = new
JTextField("000000");

  private JButton getDataButton = new JButton(
                        "Get and Display Red Color
Value");
  private String fileName = "Prob12.jpg";
  //-----------------------------------------------
------//

  public Prob12Runner(){//constructor

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.RED);
    colorPanel.add(new JLabel(
                          "Red pixel Color at
Cursor: "));
    colorPanel.add(redField);

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(getDataButton);
```

```
    getContentPane().add(controlPanel);
    setTitle("Dick Baldwin");

    setVisible(true);

    //Create a Picture object and add a name to
    // the picture before using it to construct
the
    // PictureExplorer object.
    pix = new Picture(fileName);
    pix.addMessage("Dick Baldwin",10,20);

    //Create a PictureExplorer object containing
the
    // picture and set its default close
operation.
    explorer = new PictureExplorer(pix);
    //Get a ref to the JFrame in the
PictureExplorer
    // object.
    explorerFrame = explorer.getFrame();
    explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);

    //Set the size of the control GUI.
    setSize(explorerFrame.getWidth(),110);

    //Set the location for the control GUI
immediately
    // below the PictureExplorer object, and set
its
    // default close operation.
    setLocation(0,explorerFrame.getHeight());
    setDefaultCloseOperation(
```

```
WindowConstants.EXIT_ON_CLOSE);

    //-----------------------------------------
------//
    //Register a listener object on the button.
    //-----------------------------------------
------//
    getDataButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{

          redField.setText(explorer.getRValue());

        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
------//

  }//end constructor
}//end class Prob12Runner
```

Listing 11 - Modified PictureExplorer.java.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;

/*
05/15/12
Modified by Baldwin to add getter methods to cause
the following values to be accessible from outside
the
PictureExplorer object:
```

```
Red color value

Reference to the JFrame containing the

PictureExplorer
object.

Also disabled the call to setDefaultCloseOperation
*/

/**
 * Displays a picture and lets you explore the
picture by displaying the x, y, red,
 * green, and blue values of the pixel at the
cursor when you click a mouse button or
 * press and hold a mouse button while moving the
cursor.  It also lets you zoom in or
 * out.  You can also type in a x and y value to
see the color at that location.
 *
 * Originally created for the Jython Environment
for Students (JES).
 * Modified to work with DrJava by Barbara Ericson
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Keith McDermottt, gte047w@cc.gatech.edu
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class PictureExplorer implements
MouseMotionListener, ActionListener, MouseListener
{

 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;

 //Main gui variables
 private JFrame pictureFrame;
```

```
private JScrollPane scrollPane;

//information bar variables

private JLabel xLabel;
private JButton xPrevButton;
private JButton yPrevButton;
private JButton xNextButton;
private JButton yNextButton;
private JLabel yLabel;
private JTextField xValue;
private JTextField yValue;
private JLabel rValue;
private JLabel gValue;
private JLabel bValue;
private JLabel colorLabel;
private JPanel colorPanel;

// menu components
private JMenuBar menuBar;
private JMenu zoomMenu;
private JMenuItem twentyFive;
private JMenuItem fifty;
private JMenuItem seventyFive;
private JMenuItem hundred;
private JMenuItem hundredFifty;
private JMenuItem twoHundred;
private JMenuItem fiveHundred;

/** The picture being explored */
private DigitalPicture picture;

/** The image icon used to display the picture */
private ImageIcon scrollImageIcon;

/** The image display */
private ImageDisplay imageDisplay;
```

```
 /** the zoom factor (amount to zoom) */
 private double zoomFactor;


 /** the number system to use, 0 means starting at
0, 1 means starting at 1 */
 private int numberBase=0;

 /**
  * Public constructor
  * @param picture the picture to explore
  */
 public PictureExplorer(DigitalPicture picture)
 {
   // set the fields
   this.picture=picture;
   zoomFactor=1;

   // create the window and set things up
   createWindow();
 }

//===Methods added by Baldwin on
05/15/12==============//
  //Method to get the red color value as text.
  public String getRValue(){
    return rValue.getText();
  }//end getRValue

  //Method to get a reference to the JFrame
containing
  // the PictureExplorer object.
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()
//===End methods added by Baldwin on
05/15/12==========//
```

```
 /**
  * Changes the number system to start at one
  */
 public void changeToBaseOne()
 {
   numberBase=1;
 }

 /**
  * Set the title of the frame
  *@param title the title to use in the JFrame
  */
 public void setTitle(String title)
 {
   pictureFrame.setTitle(title);
 }

 /**
  * Method to create and initialize the picture
frame
  */
 private void createAndInitPictureFrame()
 {
   pictureFrame = new JFrame(); // create the
JFrame
   pictureFrame.setResizable(true);  // allow the
user to resize it
   pictureFrame.getContentPane().setLayout(new
BorderLayout()); // use border layout

   /*Disabled by Baldwin on 5/15/12
   pictureFrame.setDefaultCloseOperation(
```

```
   .                                .          `
JFrame.DISPOSE_ON_CLOSE);
   */


   pictureFrame.setTitle(picture.getTitle());
   PictureExplorerFocusTraversalPolicy newPolicy =
new PictureExplorerFocusTraversalPolicy();

pictureFrame.setFocusTraversalPolicy(newPolicy);

 }

 /**
  * Method to create the menu bar, menus, and menu
items
  */
 private void setUpMenuBar()
 {
   //create menu
   menuBar = new JMenuBar();
   zoomMenu = new JMenu("Zoom");
   twentyFive = new JMenuItem("25%");
   fifty = new JMenuItem("50%");
   seventyFive = new JMenuItem("75%");
   hundred = new JMenuItem("100%");
   hundred.setEnabled(false);
   hundredFifty = new JMenuItem("150%");
   twoHundred = new JMenuItem("200%");
   fiveHundred = new JMenuItem("500%");

   // add the action listeners
   twentyFive.addActionListener(this);
   fifty.addActionListener(this);
   seventyFive.addActionListener(this);
   hundred.addActionListener(this);
   hundredFifty.addActionListener(this);
   twoHundred.addActionListener(this);
```

```
    fiveHundred.addActionListener(this);

    // add the menu items to the menus
    zoomMenu.add(twentyFive);
    zoomMenu.add(fifty);
    zoomMenu.add(seventyFive);
    zoomMenu.add(hundred);
    zoomMenu.add(hundredFifty);
    zoomMenu.add(twoHundred);
    zoomMenu.add(fiveHundred);
    menuBar.add(zoomMenu);

    // set the menu bar to this menu
    pictureFrame.setJMenuBar(menuBar);
  }

  /**
   * Create and initialize the scrolling image
   */
  private void createAndInitScrollingImage()
  {
    scrollPane = new JScrollPane();

    BufferedImage bimg =
picture.getBufferedImage();
    imageDisplay = new ImageDisplay(bimg);
    imageDisplay.addMouseMotionListener(this);
    imageDisplay.addMouseListener(this);
    imageDisplay.setToolTipText("Click a mouse
button on a pixel to see the pixel information");
    scrollPane.setViewportView(imageDisplay);
    pictureFrame.getContentPane().add(scrollPane,
BorderLayout.CENTER);
  }

  /**
   * Creates the JFrame and sets everything up
```

```
  */
 private void createWindow()
 {
   // create the picture frame and initialize it
   createAndInitPictureFrame();

   // set up the menu bar
   setUpMenuBar();

   //create the information panel
   createInfoPanel();

   //creates the scrollpane for the picture
   createAndInitScrollingImage();

   // show the picture in the frame at the size it
needs to be
   pictureFrame.pack();
   pictureFrame.setVisible(true);
 }

 /**
  * Method to set up the next and previous buttons
for the
  * pixel location information
  */
 private void setUpNextAndPreviousButtons()
 {
   // create the image icons for the buttons
   Icon prevIcon = new
ImageIcon(SoundExplorer.class.getResource("leftArr
ow.gif"),
                                  "previous
index");
   Icon nextIcon = new
ImageIcon(SoundExplorer.class.getResource("rightAr
row.gif"),
```

```
                                        "next index");
   // create the arrow buttons
   xPrevButton = new JButton(prevIcon);

   xNextButton = new JButton(nextIcon);
   yPrevButton = new JButton(prevIcon);
   yNextButton = new JButton(nextIcon);

   // set the tool tip text
   xNextButton.setToolTipText("Click to go to the
next x value");
   xPrevButton.setToolTipText("Click to go to the
previous x value");
   yNextButton.setToolTipText("Click to go to the
next y value");
   yPrevButton.setToolTipText("Click to go to the
previous y value");

   // set the sizes of the buttons
   int prevWidth = prevIcon.getIconWidth() + 2;
   int nextWidth = nextIcon.getIconWidth() + 2;
   int prevHeight = prevIcon.getIconHeight() + 2;
   int nextHeight = nextIcon.getIconHeight() + 2;
   Dimension prevDimension = new
Dimension(prevWidth,prevHeight);
   Dimension nextDimension = new
Dimension(nextWidth, nextHeight);
   xPrevButton.setPreferredSize(prevDimension);
   yPrevButton.setPreferredSize(prevDimension);
   xNextButton.setPreferredSize(nextDimension);
   yNextButton.setPreferredSize(nextDimension);

   // handle previous x button press
   xPrevButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt)
{
       xIndex--;
```

```
        if (xIndex < 0)
          xIndex = 0;
        displayPixelInformation(xIndex,yIndex);

      }
    });

    // handle previous y button press
    yPrevButton.addActionListener(new
ActionListener() {
      public void actionPerformed(ActionEvent evt)
{
        yIndex--;
        if (yIndex < 0)
          yIndex = 0;
        displayPixelInformation(xIndex,yIndex);
      }
    });

    // handle next x button press
    xNextButton.addActionListener(new
ActionListener() {
      public void actionPerformed(ActionEvent evt)
{
        xIndex++;
        if (xIndex >= picture.getWidth())
          xIndex = picture.getWidth() - 1;
        displayPixelInformation(xIndex,yIndex);
      }
    });

    // handle next y button press
    yNextButton.addActionListener(new
ActionListener() {
      public void actionPerformed(ActionEvent evt)
{
        yIndex++;
        if (yIndex >= picture.getHeight())
```

```
          yIndex = picture.getHeight() - 1;
        displayPixelInformation(xIndex,yIndex);
      }

    });
  }

  /**
   * Create the pixel location panel
   * @param labelFont the font for the labels
   * @return the location panel
   */
  public JPanel createLocationPanel(Font labelFont)
{

    // create a location panel
    JPanel locationPanel = new JPanel();
    locationPanel.setLayout(new FlowLayout());
    Box hBox = Box.createHorizontalBox();

    // create the labels
    xLabel = new JLabel("X:");
    yLabel = new JLabel("Y:");

    // create the text fields
    xValue = new JTextField(Integer.toString(xIndex
+ numberBase),6);
    xValue.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {

displayPixelInformation(xValue.getText(),yValue.ge
tText());
      }
    });
    yValue = new JTextField(Integer.toString(yIndex
+ numberBase),6);
    yValue.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

```
displayPixelInformation(xValue.getText(),yValue.ge
tText());
      }
    });

    // set up the next and previous buttons
    setUpNextAndPreviousButtons();

    // set up the font for the labels
    xLabel.setFont(labelFont);
    yLabel.setFont(labelFont);
    xValue.setFont(labelFont);
    yValue.setFont(labelFont);

    // add the items to the vertical box and the
box to the panel
    hBox.add(Box.createHorizontalGlue());
    hBox.add(xLabel);
    hBox.add(xPrevButton);
    hBox.add(xValue);
    hBox.add(xNextButton);
    hBox.add(Box.createHorizontalStrut(10));
    hBox.add(yLabel);
    hBox.add(yPrevButton);
    hBox.add(yValue);
    hBox.add(yNextButton);
    locationPanel.add(hBox);
    hBox.add(Box.createHorizontalGlue());

    return locationPanel;
  }

  /**
   * Create the color information panel
   * @param labelFont the font to use for labels
   * @return the color information panel
```

```
  */

 private JPanel createColorInfoPanel(Font
labelFont)
 {
   // create a color info panel
   JPanel colorInfoPanel = new JPanel();
   colorInfoPanel.setLayout(new FlowLayout());

   // get the pixel at the x and y
   Pixel pixel = new Pixel(picture,xIndex,yIndex);

   // create the labels
   rValue = new JLabel("R: " + pixel.getRed());
   gValue = new JLabel("G: " + pixel.getGreen());
   bValue = new JLabel("B: " + pixel.getBlue());

   // create the sample color panel and label
   colorLabel = new JLabel("Color at location: ");
   colorPanel = new JPanel();
   colorPanel.setBorder(new
LineBorder(Color.black,1));

   // set the color sample to the pixel color
   colorPanel.setBackground(pixel.getColor());

   // set the font
   rValue.setFont(labelFont);
   gValue.setFont(labelFont);
   bValue.setFont(labelFont);
   colorLabel.setFont(labelFont);
   colorPanel.setPreferredSize(new
Dimension(25,25));

   // add items to the color information panel
   colorInfoPanel.add(rValue);
   colorInfoPanel.add(gValue);
```

```
    colorInfoPanel.add(bValue);
    colorInfoPanel.add(colorLabel);
    colorInfoPanel.add(colorPanel);


    return colorInfoPanel;
  }

  /**
   * Creates the North JPanel with all the pixel
location
   * and color information
   */
  private void createInfoPanel()
  {
    // create the info panel and set the layout
    JPanel infoPanel = new JPanel();
    infoPanel.setLayout(new BorderLayout());

    // create the font
    Font largerFont = new
Font(infoPanel.getFont().getName(),

infoPanel.getFont().getStyle(),14);

    // create the pixel location panel
    JPanel locationPanel =
createLocationPanel(largerFont);

    // create the color informaiton panel
    JPanel colorInfoPanel =
createColorInfoPanel(largerFont);

    // add the panels to the info panel

infoPanel.add(BorderLayout.NORTH,locationPanel);

infoPanel.add(BorderLayout.SOUTH,colorInfoPanel);
```

```
   // add the info panel


pictureFrame.getContentPane().add(BorderLayout.NOR
TH,infoPanel);
 }

 /**
  * Method to check that the current position is
in the viewing area and if
  * not scroll to center the current position if
possible
  */
 public void checkScroll()
 {
   // get the x and y position in pixels
   int xPos = (int) (xIndex * zoomFactor);
   int yPos = (int) (yIndex * zoomFactor);

   // only do this if the image is larger than
normal
   if (zoomFactor > 1) {

     // get the rectangle that defines the current
view
     JViewport viewport =
scrollPane.getViewport();
     Rectangle rect = viewport.getViewRect();
     int rectMinX = (int) rect.getX();
     int rectWidth = (int) rect.getWidth();
     int rectMaxX = rectMinX + rectWidth - 1;
     int rectMinY = (int) rect.getY();
     int rectHeight = (int) rect.getHeight();
     int rectMaxY = rectMinY + rectHeight - 1;

     // get the maximum possible x and y index
     int maxIndexX = (int) (picture.getWidth() *
```

```
zoomFactor) - rectWidth - 1;
     int maxIndexY = (int) (picture.getHeight() *
zoomFactor) - rectHeight - 1;


     // calculate how to position the current
position in the middle of the viewing
     // area
     int viewX = xPos - (int) (rectWidth / 2);
     int viewY = yPos - (int) (rectHeight / 2);

     // reposition the viewX and viewY if outside
allowed values
     if (viewX < 0)
       viewX = 0;
     else if (viewX > maxIndexX)
       viewX = maxIndexX;
     if (viewY < 0)
       viewY = 0;
     else if (viewY > maxIndexY)
       viewY = maxIndexY;

     // move the viewport upper left point
     viewport.scrollRectToVisible(new
Rectangle(viewX,viewY,rectWidth,rectHeight));
   }
 }

 /**
  * Zooms in the on picture by scaling the image.
  * It is extremely memory intensive.
  * @param factor the amount to zoom by
  */
 public void zoom(double factor)
 {
   // save the current zoom factor
   zoomFactor = factor;
```

```
   // calculate the new width and height and get
an image that size
   int width = (int)
(picture.getWidth()*zoomFactor);
   int height = (int)
(picture.getHeight()*zoomFactor);
   BufferedImage bimg =
picture.getBufferedImage();

   // set the scroll image icon to the new image

imageDisplay.setImage(bimg.getScaledInstance(width
, height, Image.SCALE_DEFAULT));
   imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
   imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
   imageDisplay.revalidate();
   checkScroll();  // check if need to reposition
scroll
 }

 /**
  * Repaints the image on the scrollpane.
  */
 public void repaint()
 {
   pictureFrame.repaint();
 }

 //****************************************//
 //                Event Listeners          //
 //****************************************//

 /**
  * Called when the mouse is dragged (button held
down and moved)
```

```
  * @param e the mouse event
  */
 public void mouseDragged(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method to check if the given x and y are in
the picture
  * @param x the horiztonal value
  * @param y the vertical value
  * @return true if the x and y are in the picture
and false otherwise
  */
 private boolean isLocationInPicture(int x, int y)
 {
   boolean result = false; // the default is false
   if (x >= 0 && x < picture.getWidth() &&
       y >= 0 && y < picture.getHeight())
     result = true;

   return result;
 }

 /**
  * Method to display the pixel information from
the passed x and y but
  * also converts x and y from strings
  * @param xString the x value as a string from
the user
  * @param yString the y value as a string from
the user
  */
 public void displayPixelInformation(String
xString, String yString)
 {
```

```
    int x = -1;
    int y = -1;
    try {
      x = Integer.parseInt(xString);
      x = x - numberBase;
      y = Integer.parseInt(yString);
      y = y - numberBase;
    } catch (Exception ex) {
    }

    if (x >= 0 && y >= 0) {
      displayPixelInformation(x,y);
    }
  }

  /**
   * Method to display pixel information for the
passed x and y
   * @param pictureX the x value in the picture
   * @param pictureY the y value in the picture
   */
  private void displayPixelInformation(int
pictureX, int pictureY)
  {
    // check that this x and y is in range
    if (isLocationInPicture(pictureX, pictureY))
    {
      // save the current x and y index
      xIndex = pictureX;
      yIndex = pictureY;

      // get the pixel at the x and y
      Pixel pixel = new
Pixel(picture,xIndex,yIndex);

      // set the values based on the pixel
      xValue.setText(Integer.toString(xIndex  +
```

```
numberBase));
     yValue.setText(Integer.toString(yIndex +
numberBase));

     rValue.setText("R: " + pixel.getRed());
     gValue.setText("G: " + pixel.getGreen());
     bValue.setText("B: " + pixel.getBlue());
     colorPanel.setBackground(new
Color(pixel.getRed(), pixel.getGreen(),
pixel.getBlue()));

   }
   else
   {
     clearInformation();
   }

   // notify the image display of the current x
and y
   imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
   imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
 }

 /**
  * Method to display pixel information based on a
mouse event
  * @param e a mouse event
  */
 private void displayPixelInformation(MouseEvent
e)
 {

   // get the cursor x and y
   int cursorX = e.getX();
   int cursorY = e.getY();
```

```
   // get the x and y in the original (not scaled
image)
   int pictureX = (int) (cursorX / zoomFactor +
numberBase);
   int pictureY = (int) (cursorY / zoomFactor +
numberBase);

   // display the information for this x and y
   displayPixelInformation(pictureX,pictureY);

 }

 /**
  * Method to clear the labels and current color
and reset the
  * current index to -1
  */
 private void clearInformation()
 {
   xValue.setText("N/A");
   yValue.setText("N/A");
   rValue.setText("R: N/A");
   gValue.setText("G: N/A");
   bValue.setText("B: N/A");
   colorPanel.setBackground(Color.black);
   xIndex = -1;
   yIndex = -1;
 }

 /**
  * Method called when the mouse is moved with no
buttons down
  * @param e the mouse event
  */
 public void mouseMoved(MouseEvent e)
 {}
```

```
 /**
  * Method called when the mouse is clicked
  * @param e the mouse event
  */
 public void mouseClicked(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is pushed
down
  * @param e the mouse event
  */
 public void mousePressed(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is
released
  * @param e the mouse event
  */
 public void mouseReleased(MouseEvent e)
 {
 }

 /**
  * Method called when the component is entered
(mouse moves over it)
  * @param e the mouse event
  */
 public void mouseEntered(MouseEvent e)
 {
 }
```

```
 /**
  * Method called when the mouse moves over the
component
  * @param e the mouse event
  */
 public void mouseExited(MouseEvent e)
 {
 }

 /**
  * Method to enable all menu commands
  */
 private void enableZoomItems()
 {
   twentyFive.setEnabled(true);
   fifty.setEnabled(true);
   seventyFive.setEnabled(true);
   hundred.setEnabled(true);
   hundredFifty.setEnabled(true);
   twoHundred.setEnabled(true);
   fiveHundred.setEnabled(true);
 }

 /**
  * Controls the zoom menu bar
  *
  * @param a the ActionEvent
  */
 public void actionPerformed(ActionEvent a)
 {

   if(a.getActionCommand().equals("Update"))
   {
     this.repaint();
   }

   if(a.getActionCommand().equals("25%"))
```

```
{
  this.zoom(.25);
  enableZoomItems();
  twentyFive.setEnabled(false);
}

if(a.getActionCommand().equals("50%"))
{
  this.zoom(.50);
  enableZoomItems();
  fifty.setEnabled(false);
}

if(a.getActionCommand().equals("75%"))
{
  this.zoom(.75);
  enableZoomItems();
  seventyFive.setEnabled(false);
}

if(a.getActionCommand().equals("100%"))
{
  this.zoom(1.0);
  enableZoomItems();
  hundred.setEnabled(false);
}

if(a.getActionCommand().equals("150%"))
{
  this.zoom(1.5);
  enableZoomItems();
  hundredFifty.setEnabled(false);
}

if(a.getActionCommand().equals("200%"))
{
  this.zoom(2.0);
```

```
      enableZoomItems();
      twoHundred.setEnabled(false);
    }


    if(a.getActionCommand().equals("500%"))
    {
      this.zoom(5.0);
      enableZoomItems();
      fiveHundred.setEnabled(false);
    }
  }

  /**
   * Test Main.  It will ask you to pick a file and
then show it
   */
  public static void main( String args[])
  {
    Picture p = new
Picture(FileChooser.pickAFile());
    PictureExplorer test = new PictureExplorer(p);

  }

  /**
   * Class for establishing the focus for the
textfields
   */
  private class PictureExplorerFocusTraversalPolicy
                  extends FocusTraversalPolicy {

        /**
         * Method to get the next component for
focus
         */
        public Component
getComponentAfter(Container focusCycleRoot,
```

```
Component aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
        }

        /**
         * Method to get the previous component
for focus
         */
         public Component
getComponentBefore(Container focusCycleRoot,
                                          Component
aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
         }

         public Component
getDefaultComponent(Container focusCycleRoot) {
            return xValue;
        }

        public Component
getLastComponent(Container focusCycleRoot) {
            return yValue;
        }

        public Component
getFirstComponent(Container focusCycleRoot) {
            return xValue;
        }
    }
```

```
}
```

-end-

Java3126 Java OOP Handling document events on a text field and creating a color swatch
Learn how to handle document events on text fields containing color values. Also learn how to create a color swatch.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output after clicking the button.
- Figure 3 . Program output after entering color values.

Listings

- Listing 1 . The driver class.
- Listing 2 . Beginning of the class named Prob13Runner.
- Listing 3 . Beginning of constructor for Prob13Runner.
- Listing 4 . Continue the constructor.
- Listing 5 . Register a listener object on the button.
- Listing 6 . Beginning of DocumentListener on Red text field.
- Listing 7 . The removeUpdate method.
- Listing 8 . The insertUpdate method.
- Listing 9 . The paintColorSwatch method.
- Listing 10 . Complete program listing.

Preview

In this module, I will show you how to handle document events on text fields containing color values.

I will also show you how to create a color swatch that displays the color indicated by the color values in the Red, Green, and Blue text fields.

Program output at startup

Figure 1 shows the program output at startup.

Note that the color values in the Red, Green, and Blue text fields are all zero, the color of the square at the right end of the Green panel is black.

Recall that black is represented by Red, Green, and Blue color values of zero.

Figure 1 - Program output at startup.



Program output after clicking the button

Figure 2 shows the program output after clicking the button labeled "Set Green Color."

Figure 2 - Program output after clicking the button.



Note that in Figure 2 ,

- the color values in the Red and Blue text fields are zero,
- the color value in the Green text field is 255,
- the square *(color swatch)* at the right end of the green panel is green with a black outline
- the square is not transparent.

Recall that pure green is represented by Red and Blue color values of zero, with a Green color value of 255.

Program output after entering color values

Figure 3 shows the program output after manually entering a value of 255 in the Red text field.

Note that

- the color values in the Red and Green text fields are 255,
- the color value in the Blue text field is 0, and

- the square at the right end of the green panel is yellow with a black outline.

Figure 3 - Program output after entering color values.



Recall that pure yellow is represented by Red and Green color values of 255, with a Blue color value of zero.

Color of square tracks color values in text fields

The square *(color swatch)* always has a black outline as shown in Figure 3 .

The color in the swatch always tracks the color values in the text fields regardless of whether those values are changed manually, or they are changed by program code.

Discussion and sample code

A complete listing of the program discussed in this module is provided in Listing 10 .

Will explain in fragments

I will break this program down and explain it in fragments.

The driver class

The driver class named **Prob13** is shown in Listing 1 . There is nothing new here.

Listing 1 - The driver class.

```
public class Prob13{
  public static void main(String[] args){
    new Prob13Runner();
  }//end main method
}//end class Prob13
```

Beginning of the class named Prob13Runner

The class named **Prob13Runner** begins in Listing 2 , which declares several instance variables and initializes some of them.

Listing 2 - Beginning of the class named Prob13Runner .

Listing 2 - Beginning of the class named Prob13Runner .

```
class Prob13Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new
JPanel();

  private JTextField redField = new
JTextField("000000");
  private JTextField greenField =
                                  new
JTextField("000000");
  private JTextField blueField = new
JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  private JButton setColorButton =
                             new JButton("Set
Green Color");
```

Beginning of constructor for Prob13Runner

The constructor begins in Listing 3 . The last two statements in Listing 3 may be new to you.

The **JPanel** referred to by **colorIndicatorPanel** is the color swatch shown in Figure 3 .

The last two statements in Listing 3 control the border color, and the size of the **JPanel** *(color swatch)* object.

Listing 3 - Beginning of constructor for Prob13Runner.

Listing 3 - Beginning of constructor for Prob13Runner.

```
  public Prob13Runner(){//constructor

    setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);

    controlPanel.setLayout(new
GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    colorIndicatorPanel.setBorder(
                              new
LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                    new
Dimension(20,20));
```

The constructor continues in Listing 4 .

There is nothing new in Listing 4 with the possible exception of the call to the method named **paintColorSwatch** .

I will explain the method named **paintColorSwatch** later.

Listing 4 - Continue the constructor.

```
    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(setColorButton);

    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content
pane, adjust to
    // the correct size, and set the title.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);
```

Register listeners on the user input components

This program registers event listener objects on four input components:

One component is the button at the bottom of Figure 3 labeled **Set Green Color.**

The other three components are the three text fields labeled **Red** , **Green** , and **Blue** shown in Figure 3 .

The three event listeners that are registered on the text fields are very similar. Therefore, I will explain only one of them. You can view the code for the other two in Listing 10 .

Register a listener object on the button

Listing 5 registers an ActionListener object on the **JButton** object referred to by **setColorButton** .

You have seen code like this in numerous previous modules. Therefore, a detailed explanation of Listing 5 should not be required here.

Listing 5 - Register a listener object on the button.

```
    setColorButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          //Set the color to green.
          redInt = 0;
          greenInt = 255;
          blueInt = 0;

          //Show the color values in the text
fields.
          redField.setText("" + redInt);
          greenField.setText("" + greenInt);
          blueField.setText("" + blueInt);

        }//end action performed
      }//end new ActionListener
    );//end addActionListener
```

The DocumentListener interface

The **DocumentListener** interface can be described briefly as follows:

"Interface for an observer to register to receive notifications of changes to a text document."

This interface is implemented by numerous classes including the **JTextField** class. As mentioned earlier, the **Red** , **Green** , and **Blue** text fields in Figure 3 are objects of the **JTextField** class and therefore implement the **DocumentListener** interface.

DocumentListener methods

The **DocumentListener** interface declares the follow three methods:

- **changedUpdate**
    - Gives notification that an attribute or set of attributes changed.
- **insertUpdate**
    - Gives notification that there was an insert into the document.
- **removeUpdate**
    - Gives notification that a portion of the document has been removed.

Only the **insertUpdate** and **removeUpdate** methods are of interest in this program. Therefore, the **changedUpdate** method will be implemented as an empty method.

Beginning of DocumentListener

Listing 6 shows the beginning of the definition, instantiation, and registration of a **DocumentListener** object on the object of type **Document** that is encapsulated in the Red text field.

Listing 6 - Beginning of DocumentListener on Red text field.

```
redField.getDocument().addDocumentListener(
      new DocumentListener(){

        public void
changedUpdate(DocumentEvent e){
          //Empty method - not needed
        }//end changedUpdate
```

Note that registration of the listener object in this case has an additional level of indirection *(getDocument)* as compared to the registration of the listener on the JButton in Listing 5 . In other words, the listener is not registered on the text field. Instead, it is registered on the document encapsulated in the text field.

This listener will respond when the contents of the text field are modified, either by the program, or by the user.

*(As explained earlier, the **changedUpdate** method is defined as an empty method.)*

The removeUpdate method

Listing 7 shows the **removeUpdate** method in its entirety.

Listing 7 - The removeUpdate method.

Listing 7 - The removeUpdate method.

```
        public void removeUpdate(DocumentEvent
e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <=
255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate
```

Behavior of the removeUpdate method

Listing 7 calls the static **parseInt** method of the **Integer** class to convert the **String** contents of the Red text field into a value of type **int** .

The **parseInt** method can throw a **NumberFormatException** if the string does not contain a parsable integer.

The possibility of a **NumberFormatException** , *(which is a checked exception)* requires that the call to the **parseInt** method be enclosed in a **try-catch** block.

In this program, the **catch** block causes the **removeUpdate** method to ***do nothing*** if the contents of the text field cannot be converted into an **int** , which causes a **NumberFormatException** to be thrown

If a **NumberFormatException** exception is not thrown, the return value from the **parseInt** method is stored in an instance variable named **redInt** .

If the value of **redInt** is between 0 and 255 inclusive, Listing 7 calls the method named **paintColorSwatch** , causing the color swatch to be repainted, using the **Red** value, along with the **Green** and **Blue** values computed elsewhere.

The insertUpdate method

The **insertUpdate** method is shown in Listing 8 .

The behavior of the **insertUpdate** method is essentially the same as the behavior of the **removeUpdate** method explained earlier. Therefore, an explanation of the **insertUpdate** method should not be needed.

Listing 8 - The insertUpdate method.

Listing 8 - The insertUpdate method.

```
        public void insertUpdate(DocumentEvent
e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <=
255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate
        //-----------------------------------
-------//

      }//end new DocumentListener
    );//end addDocumentListener
```

The end of the class definition

Listing 8 also signals the end of the anonymous class definition that began in Listing 6 .

The Green and Blue text fields

The Green and Blue text fields are processed using very similar **DocumentListener** objects as used for the Red text field. You can view that code in Listing 10 .

That ends the discussion of the constructor for the class named **Prob13Runner** .

The method named paintColorSwatch

There were earlier references to a method named **paintColorSwatch** . The code for the **paintColorSwatch** method is shown in Listing 9 .

Listing 9 - The paintColorSwatch method.

```
  private void paintColorSwatch(){
    colorIndicatorPanel.setBackground(
                        new
Color(redInt,greenInt,blueInt));
  }//end paintColorSwatch
  //-----------------------------------------
----------//

}//end class Prob13Runner
```

The **paintColorSwatch** method sets the background color for the **JPanel** object that represents the color swatch, using values of red, green, and blue that are computed elsewhere in the program.

The end of the program

Listing 9 also signals the end of the class named **Prob13Runner** and the end of the program.

Run the program

I encourage you to copy the code from Listing 10 . Compile the code and execute it. Experiment with the code, making changes, and observing the

results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this lesson, you learned how to handle document events on text fields containing color values. You also learned how to create a color swatch.

What's next?

In the next module, you will learn how to use a JColorChooser object to specify a color in any one of five different ways.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Handling document events on a text field and creating a color swatch
- File: Java1326.htm
- Published: 09/11/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the source code discussed in this module is shown in Listing 10 .

Listing 10 - Complete program listing.

```
/*File Prob13 Copyright 2012 R.G.Baldwin

This program handles document events on the
contents of
text fields containing color values.

The values are used to create a color swatch that
displays
the color indicated by the color values in the
red, green,
and blue text fields.
**************************************************
*******/
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
```

```
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;

public class Prob13{
  public static void main(String[] args){
    new Prob13Runner();
  }//end main method
}//end class Prob13
//===========================================
======//

class Prob13Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new
JPanel();

  private JTextField redField = new
JTextField("000000");
  private JTextField greenField =
                                    new
JTextField("000000");
  private JTextField blueField = new
JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
```

```
  private int blueInt = 0;

  private JButton setColorButton =
                             new JButton("Set Green
Color");
  //-----------------------------------------------
------//

  public Prob13Runner(){//constructor

    setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    colorIndicatorPanel.setBorder(
                             new
LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                    new
Dimension(20,20));

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(setColorButton);
```

```
    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content pane,
adjust to
    // the correct size, and set the title.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);

    //-----------------------------------------
------//
    //Register listeners on the user input
components.
    //-----------------------------------------
------//
    setColorButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          //Set the color to green.
          redInt = 0;
          greenInt = 255;
          blueInt = 0;

          //Show the color values in the text
fields.
          redField.setText("" + redInt);
          greenField.setText("" + greenInt);
          blueField.setText("" + blueInt);

        }//end action performed
      }//end new ActionListener
    );//end addActionListener
```

```
    //-----------------------------------------
------//

    //Register a document listener on the red text
field.
    // This listener will respond when the
contents of
    // the text field are modified either by the
program
    // or by the user.
    redField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{
          //Empty method - not needed
        }//end changedUpdate

        public void removeUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <= 255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <= 255)){
              paintColorSwatch();
```

```
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the green
text
    // field. Essentially the same as the above.
    greenField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            greenInt = Integer.parseInt(

greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            greenInt = Integer.parseInt(
```

```
greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the blue
text
    // field. Essentially the same as the above.
    blueField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
```

```
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-------------------------------------------
------//
  }//end constructor
  //-------------------------------------------
------//

  //The purpose of this method is to color a
swatch
  // located next to the RGB color values.
  private void paintColorSwatch(){
    colorIndicatorPanel.setBackground(
                      new
Color(redInt,greenInt,blueInt));
  }//end paintColorSwatch
  //-------------------------------------------
------//


}//end class Prob13Runner
```

-end-

Java3128 Java OOP Using a JColorChooser object
Learn how to use a JColorChooser object to specify a color in any one of five different ways.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- What's next?
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output after clicking the Choose Color button.
- Figure 3 . Program output after selecting a reddish color and clicking the OK button.
- Figure 4 . Program output after clicking the Darker button.
- Figure 5 . The showDialog method.

Listings

- Listing 1 . Event handler for the Choose Color button.
- Listing 2 . Show the JColorChooser object.
- Listing 3 . Darkening the color.
- Listing 4 . Complete program listing.

Preview

In this module, you will learn how to use a **JColorChooser** object to specify a color in any one of five different ways:

- Swatches
- HSV
- HSL
- RGB
- CMYK

The details regarding the five different ways are not explained. Instead, an understanding of the five ways for specifying a color is left as an exercise for the student.

This module concentrates on the programming aspects of the color chooser as opposed to the aesthetic aspects of the color chooser.

You will also learn how to create brighter and darker shades of a given color.

What is a JColorChooser object?

According to the Java documentation,

"JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color."

Program output at startup

A complete listing of the program discussed in this module is provided in Listing 4 near the end of the module.

Figure 1 shows the program output at startup.

The program output is a GUI containing

- three text fields, one each for red, green, and blue color values
- three labels that identify the contents of each text field
- a button labeled **Choose Color**
- two buttons labeled **Brighter** and **Darker**
- a square ***color swatch*** whose color reflects the color specified by the color component values in the text fields

Figure 1 - Program output at startup.



Program output after clicking the Choose Color button.

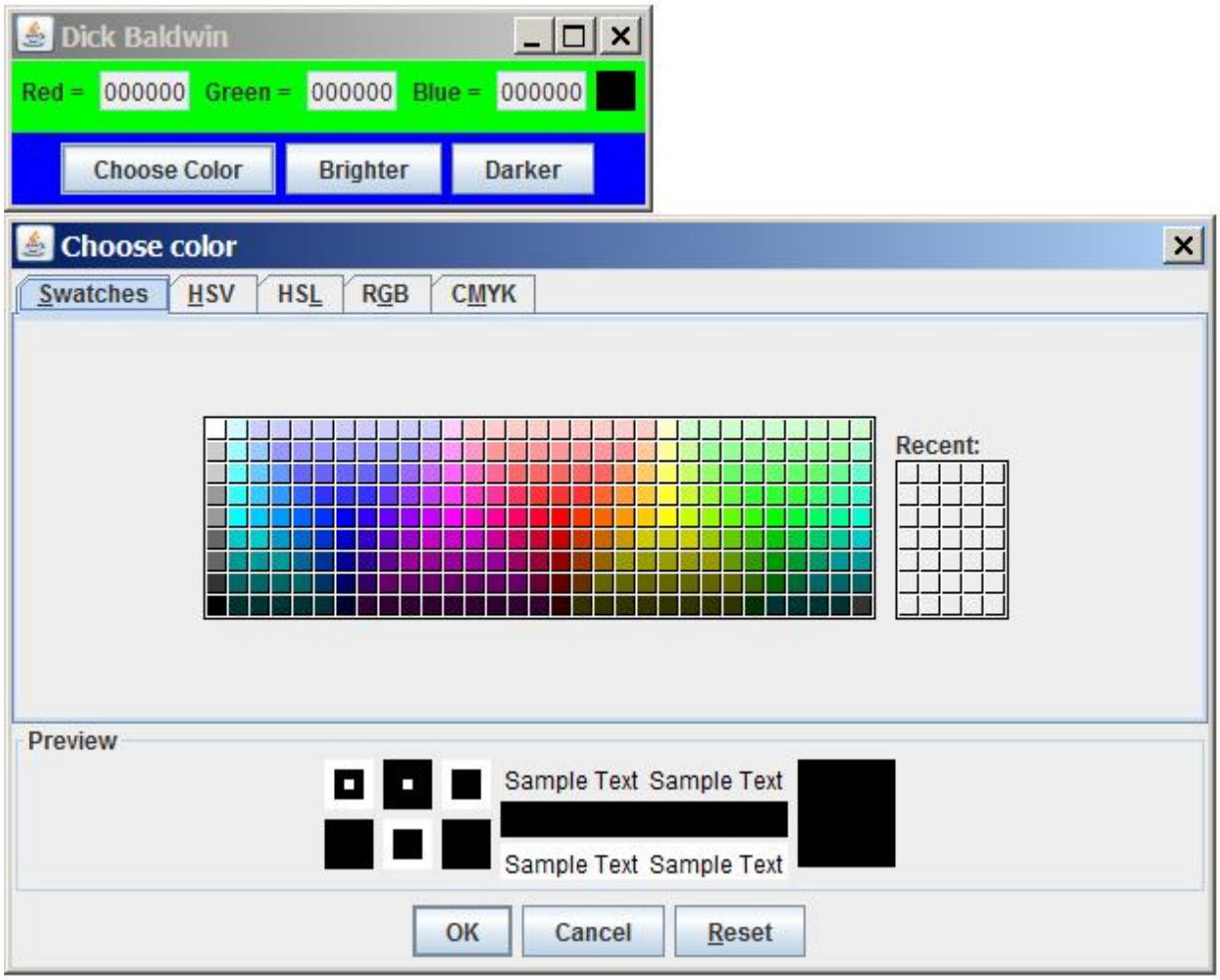
Figure 2 shows the result of clicking the **Choose Color** button shown in Figure 1 .

The bottom image in Figure 2 is an object of the **JColorChooser** class.

(The color chooser object doesn't actually appear below the GUI as shown in Figure 2 . Instead, it appears in the upper-left corner of the screen and

hides the GUI. I manually dragged it down below the GUI to produce this image.)

Figure 2 - Program output after clicking the Choose Color button.



The initial color selection

Note in Figure 2 that the selected color when the color chooser appears matches the color of the square color swatch in Figure 1 *(black)*

Program output after selecting a reddish color and clicking the OK button.

Figure 3 shows the result of

- selecting a reddish color in the color chooser, and

- clicking the OK button in the color chooser.

Figure3 - Program output after selecting a reddish color and clicking the OK button.



New color in the GUI

Note in Figure 3 that

- the color selected in the color chooser now appears in the square color swatch in Figure 3
- the color swatch in Figure 3 has a black border
- the color values in the three text fields describe the color showing in the swatch

(The color in the swatch is based on the three color values in the text fields.)

Program output after clicking the Darker button.

Figure 4 shows the result of clicking the **Darker** button once.

Figure 4 - Program output after clicking the Darker button.



A darker color

The color swatch in Figure 4 shows a darker version of the color swatch shown in Figure 3 .

The numeric color values in Figure 4 describe the color in the color swatch, and are lower than the corresponding color values in Figure 3

A brighter color

Although not demonstrated here, the color in the swatch could be made brighter by clicking the **Brighter** button.

If you click the **Darker** button enough times, the color will go to black. Similarly, if you click the **Brighter** button enough times, the color will go to white.

Discussion and sample code

A complete listing of the program is provided in Listing 4 .

Much of the code in this program is very similar to code that I explained in earlier modules. Therefore, I won't repeat those explanations here. Instead, I will concentrate on the code that is new and different.

Given that caveat, I will skip all the way down to the event handler for the **Choose Color** button.

Event handler for the Choose Color button

Listing 1 defines, instantiates, and registers an **ActionListener** object on the **Choose Color** button.

Listing 1 - Event handler for the Choose Color button.

Listing 1 - Event handler for the Choose Color button.

```
    chooseButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          Color selColor =
JColorChooser.showDialog(
                     chooseButton,"Choose
color",new Color(

redInt,greenInt,blueInt));
          if(selColor != null){
            //Don't change the color if the
user cancels
            // out.
            redField.setText("" +
selColor.getRed());
            greenField.setText("" +
selColor.getGreen());
            blueField.setText("" +
selColor.getBlue());
          }//end if

        }//end action performed
      }//end new ActionListener
    );//end addActionListener
```

Show the JColorChooser object

Listing 2 shows the code, *(extracted from Listing 1)* that causes the **color chooser** to appear on the screen.

Listing 2 - Show the JColorChooser object.

```
Color selColor = JColorChooser.showDialog(
                    chooseButton,
                    "Choose color",
                    new
Color(redInt,greenInt,blueInt));
```

Call the static showDialog method

Listing 2 calls the static **showDialog** method of the **JColorChooser** class.

Figure 5 provides information for the **showDialog** method.

Figure 5 - The showDialog method.

Figure 5 - The showDialog method.

```
public static Color showDialog(
                        Component component,
                        String title,
                        Color initialColor)
                          throws
HeadlessException

Shows a modal color-chooser dialog and blocks
until the dialog is hidden. If the user
presses
the "OK" button, then this method
hides/disposes
the dialog and returns the selected color. If
the user presses the "Cancel" button or closes
the dialog without pressing "OK", then this
method hides/disposes the dialog and returns
null.

Parameters:
  component - the parent Component for the
dialog
  title - the String containing the dialog's
title
  initialColor - the initial Color set when
the
    color-chooser is shown
Returns:
  the selected color or null if the user opted
out
```

A straightforward explanation

The behavior of the dialog is explained in Figure 5 and shouldn't require further explanation.

Parameters to the showDialog method

A comparison of Listing 2 and Figure 5 shows how the color chooser is integrated into the program.

Perhaps the most important aspects of that integration are the *third parameter* and the *return value* .

The third parameter

The third parameter is an anonymous **Color** object that matches the color specified by the text fields in the GUI, which determine the color of the swatch.

Thus, the initial color for the color chooser matches the color of the swatch in the GUI when the color chooser first appears.

The return value

The return value from the color chooser is a reference to an object of type **Color** , which is saved in a variable named **selColor** .

Processing the return value

Returning to the code in Listing 1 ,

- if the return value is not null,
- the color components of the return value are converted to strings and
- stored in the red, green, and blue text fields.

What happens next?

You learned in an earlier module that if the values stored in any of the three text fields changes for any reason, **DocumentListener** objects registered on the three text fields cause the color of the swatch to change to match the new values of the color components.

Thus, the color that is chosen by the user from the color chooser modifies the values in the text fields causing the color of the swatch in the GUI to match the chosen color.

Darkening and brightening the color

Listing 3 shows an **ActionListener** object being registered on the button labeled **Darker** in Figure 1 .

Listing 3 - Darkening the color.

```
    darkerButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){
          Color color = new Color(

redInt,greenInt,blueInt).darker();
          redField.setText("" +
color.getRed());
          greenField.setText("" +
color.getGreen());
          blueField.setText("" +
color.getBlue());
        }//end action performed
      }//end newActionListener
    );//end addActionListener
```

The darkening action listener

This code creates a new **Color** object that matches the color values in the three text fields.

Then it calls the **darker** method on that **Color** object causing its color to be darkened.

Then it extracts the color components from the darker **Color** object and stores those values in the text fields.

This, in turn, causes the color swatch to change to match the new color values.

A brightening event handler

A similar event handler is registered on the button labeled **Brighter** in Figure 1 .

This code, which you can view in Listing 4 calls the **brighter** method instead of the **darker** method on the **Color** object.

End of story

That concludes the explanation of material that is new and different in this program.

Run the program

I encourage you to copy the code from Listing 4 . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you will learned how to use a **JColorChooser** object to specify a color in any one of five different ways.

You also learned how to create brighter and darker shades of a given color.

What's next?

In the next module, you will learn how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: Using a JColorChooser object
- File: Java3128.htm
- Published: 09/11/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the source code discussed in this module is shown in Listing 4 .

Listing 4 - Complete program listing.

```
/*File Prob14 Copyright 2012 R.G.Baldwin

Old material:
This program services document events on the
contents of
text fields containing color values.

The values are used to create a color swatch that
displays
the color indicated by the color values in the
red, green,
and blue text fields.

New material:
Demonstrates how to create and use a JColorChooser
dialog.
Also demonstrates use of the darker and brighter
methods.
Also demonstrates how to cause JTextField objects
to be
non-editable.
**********************************************
*******/
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;
import javax.swing.JColorChooser;

public class Prob14{
  public static void main(String[] args){
    new Prob14Runner();
  }//end main method
}//end class Prob14
//===========================================
======//

class Prob14Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new
JPanel();

  private JTextField redField = new
JTextField("000000");
  private JTextField greenField =
                                  new
JTextField("000000");
  private JTextField blueField = new
JTextField("000000");
```

```
  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  private JButton chooseButton =
                                new JButton("Choose
Color");
  private JButton brighterButton =
                                     new
JButton("Brighter");
  private JButton darkerButton = new
JButton("Darker");
  //-----------------------------------------------
------//

  public Prob14Runner(){//constructor

    setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    redField.setEditable(false);
    greenField.setEditable(false);
    blueField.setEditable(false);
```

```
    colorIndicatorPanel.setBorder(
                              new
LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                      new
Dimension(20,20));

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(chooseButton);
    buttonPanel.add(brighterButton);
    buttonPanel.add(darkerButton);

    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content pane,
adjust to
    // the correct size, and set the title.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);

    //-----------------------------------------
------//
    //Register listeners on the user input
components.
    //-----------------------------------------
------//
    chooseButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          Color selColor =
```

```
JColorChooser.showDialog(
                    chooseButton,"Choose
color",new Color(
                       redInt,greenInt,blueInt));
          if(selColor != null){
            //Don't change the color if the user
cancels
            // out.
            redField.setText("" +
selColor.getRed());
            greenField.setText(
                                "" +
selColor.getGreen());
            blueField.setText("" +
selColor.getBlue());
          }//end if

        }//end action performed
      }//end new ActionListener
    );//end addActionListener
    //-----------------------------------------
------//

    darkerButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          Color color = new Color(
               redInt,greenInt,blueInt).darker();
          redField.setText("" + color.getRed());
          greenField.setText("" +
color.getGreen());
          blueField.setText("" + color.getBlue());
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
```

```
------//

    brighterButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          Color color = new Color(
             redInt,greenInt,blueInt).brighter();
          redField.setText("" + color.getRed());
          greenField.setText("" +
color.getGreen());
          blueField.setText("" + color.getBlue());
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------
------//

    //Register a document listener on the red text
field.
    // This listener will respond when the
contents of
    // the text field are modified either by the
program
    // or by the user.
    redField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{
          //Empty method - not needed
        }//end changedUpdate

        public void removeUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
```

```
            if((redInt >= 0) && (redInt <= 255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <= 255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the green
text
    // field. Essentially the same as the above.
    greenField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            greenInt = Integer.parseInt(
```

```
greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            greenInt = Integer.parseInt(

greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the blue
text
    // field. Essentially the same as the above.
    blueField.getDocument().addDocumentListener(
      new DocumentListener(){
```

```
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-------------------------------------------
------//
  }//end constructor
  //-----------------------------------------------
```

```
------//

  //The purpose of this method is to color a
swatch
  // located next to the RGB color values.
  private void paintColorSwatch(){
    colorIndicatorPanel.setBackground(
                        new
Color(redInt,greenInt,blueInt));
  }//end paintColorSwatch

  //-----------------------------------------
------//

  //The purpose of this method is to absorb any
exceptions
  // that may be thrown by the parseInt method in
order
  // to avoid having the program abort. In the
event that
  // an exception is thrown, this method simply
returns an
  // int value of 0;
  private int goParseInt(String string){
    int result = 0;
    try{
      result = Integer.parseInt(string);
    }catch(Exception e){
      result = 0;
    }//end catch
    return result;
  }//end goParseInt

  //-----------------------------------------
------//
```

```
}//end class Prob14Runner
```

-end-

Java3130 Java OOP A Pixel Color Editor
Learn how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

Table of Contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preview
- Discussion and sample code
- Run the program
- Summary
- Miscellaneous
- Complete program listing

Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

- Figure 1 . Program output at startup.
- Figure 2 . Program output after zooming the image.
- Figure 3 . Program output after changing the pixel color to yellow.

Listings

- Listing 1 . Code for methods that were added.
- Listing 2 . Modified setDefaultCloseOperation method.
- Listing 3 . Beginning of ActionListener registered on update button.
- Listing 4 . Convert zoom factor to type String.
- Listing 5 . Set pixel color in original image.
- Listing 6 . Create and populate a new PictureExplorer object.
- Listing 7 . Call the mousePressed method.
- Listing 8 . Set the zoom state.
- Listing 9 . Complete listing of Prob15.
- Listing 10 . Complete listing of modified PictureExplorer class.

Preview

In this module, you will learn how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

Such a program could be useful, for example, for manually correcting "red eye" problems in digital images.

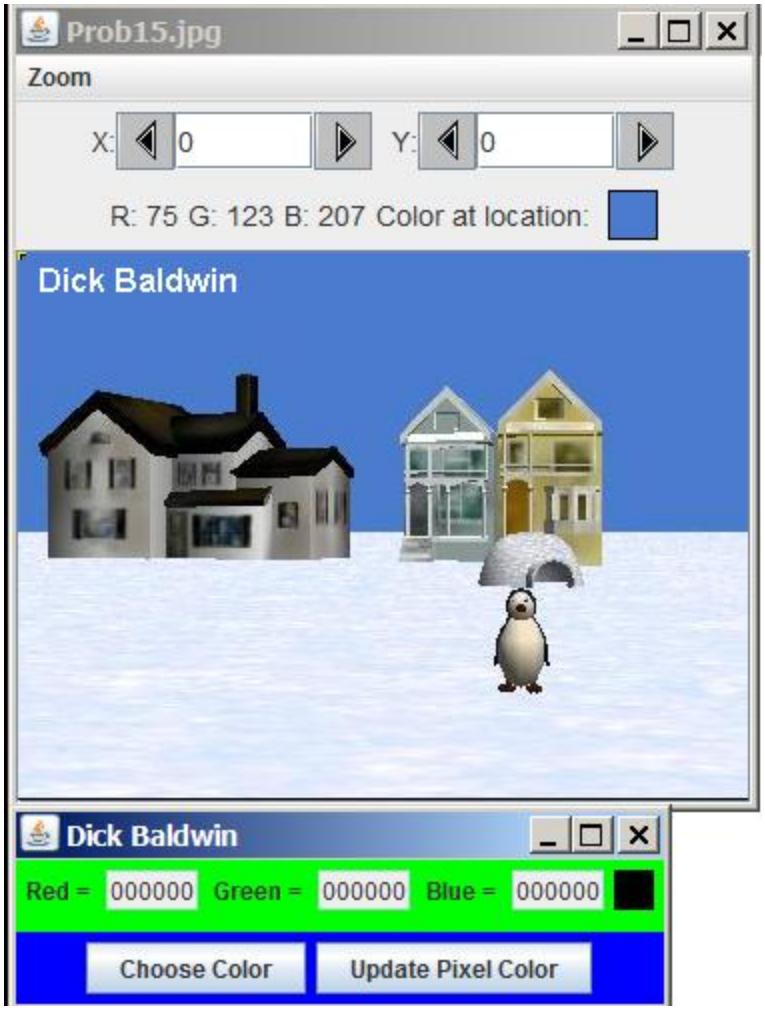
Program output at startup

Figure 1 shows the program output at startup.

The output consists of

- an Ericson **PictureExplorer** object displaying an image of a penguin, and
- a color manipulation GUI similar to those discussed in earlier modules.

Figure 1 - Program output at startup.

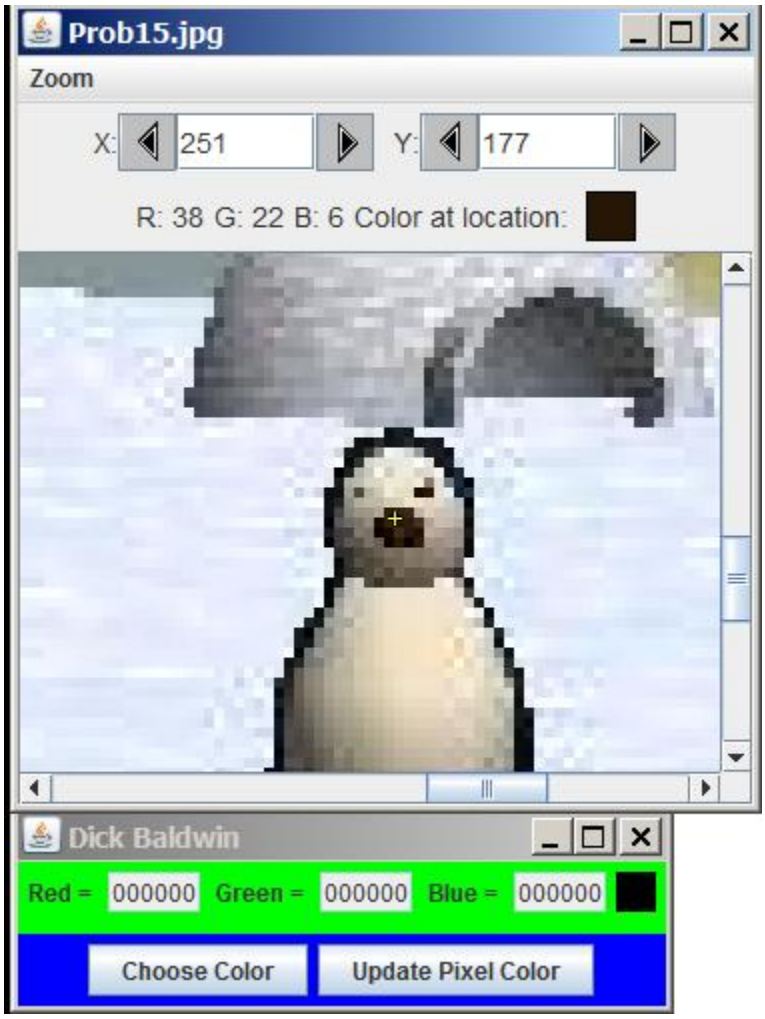


Program output after zooming the image

Figure 2 shows the program output after placing the **PictureExplorer** cursor on the penguin's nose, and zooming the **PictureExplorer** image by 500%

If you look carefully, you should be able to see the crosshair cursor on the penguin's nose.

Figure 2 - Program output after zooming the image.

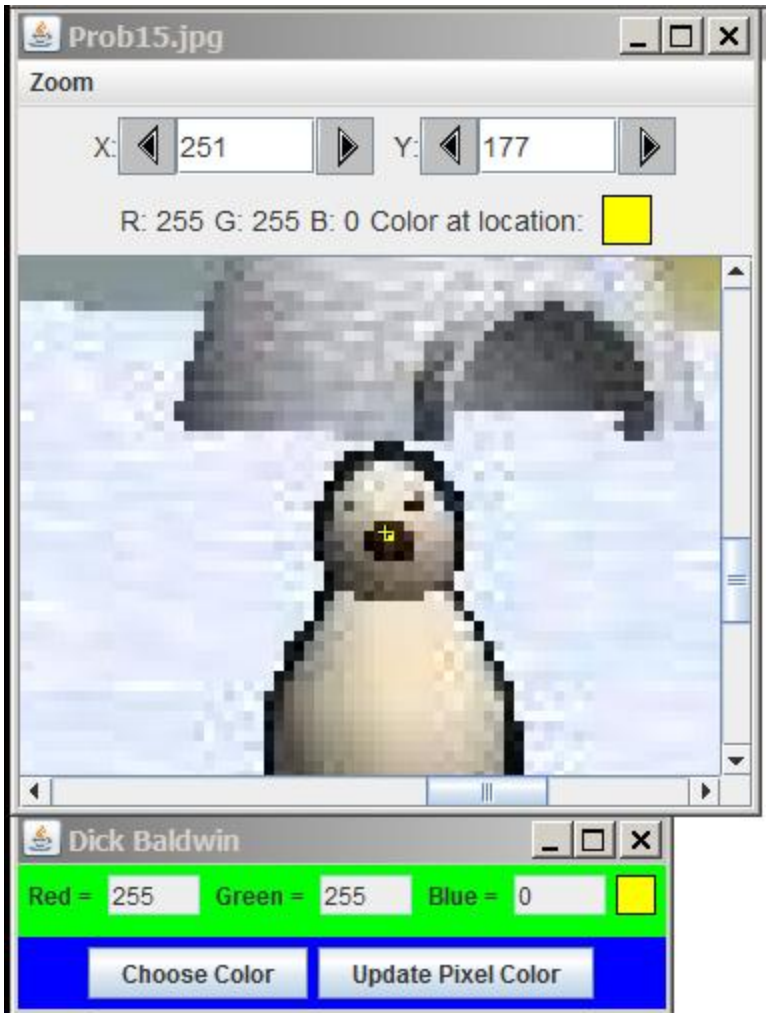


Program output after changing the pixel color to yellow

Figure 3 shows the result of selecting the color yellow with the GUI, and clicking the button labeled **Update Pixel Color**

If you look carefully, you should see that the pixel next to the crosshair cursor on the penguin's nose has changed from black to yellow.

Figure 3 - Program output after changing the pixel color to yellow.



Discussion and sample code

This program requires modifications to Ericson's **PictureExplorer** class. I will deal with that issue first.

A complete listing of the modified version of the **PictureExplorer** class is shown in Listing 10 .

Changes to Ericson's PictureExplorer class

The **PictureExplorer** class was modified to add *getter* methods to cause the following values to be accessible from outside the object:

- int xIndex

- int yIndex
- double zoomFactor
- JFrame pictureFrame

Also the call to the **setDefaultCloseOperation** method was disabled.

Code for methods that were added

Listing 1 shows the code for the methods that were added to the **PictureExplorer** class.

Listing 1 - Code for methods that were added.

Listing 1 - Code for methods that were added.

```
//Method to get the xIndex value.
public int getXIndex(){
  return xIndex;
}//end getXIndex

//Method to get the yIndex value.
public int getYIndex(){
  return yIndex;
}//end getYIndex

//Method to get the zoomFactor value.
public double getZoomFactor(){
  return zoomFactor;
}//end getZoomFactor

//Method to get a reference to the frame
public JFrame getFrame(){
  return pictureFrame;
}//end getFrame()
```

Modified setDefaultCloseOperation method

Listing 2 shows the modified version of the **setDefaultCloseOperation** method.

Note the parameter that is passed to the method when it is called.

Listing 2 - Modified setDefaultCloseOperation method.

```
explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);
```

Similar to previous code

Much of the code in this program is similar to code that I explained in earlier modules, and I won't repeat those explanations.

I will skip down to the event handler registered on the button labeled **Update Pixel Color** in Figure 1 . That code begins in Listing 3 .

Listing 3 - Beginning of ActionListener registered on update button.

```
    updateButton.addActionListener(
      new ActionListener(){
        public void
actionPerformed(ActionEvent e){

          //Get properties of PictureExplorer
object.
          xIndex = explorer.getXIndex();
          yIndex = explorer.getYIndex();
          zoomFactor =
explorer.getZoomFactor();
```

Important to preserve properties

The only thing that changes when you click the **update** button is the color of the pixel at the crosshair cursor.

Therefore, the program must preserve the state of the explorer object including such items as the zoom factor and the location of crosshair cursor.

Listing 3 gets and saves the property values that determine the state of the explorer object.

Convert zoom factor to String

The **getZoomFactor** method returns the zoom factor as type **double** . However, we need the zoom factor as type **String** . The code in Listing 4 converts the zoom factor to type **String** .

Listing 4 - Convert zoom factor to type String.

Listing 4 - Convert zoom factor to type String.

```
            //Save zoom factor as a string.
            String zoomString = "100%";
            if(zoomFactor == 0.25){
              zoomString = "25%";
            }else if(zoomFactor == 0.50){
              zoomString = "50%";
            }else if(zoomFactor == 0.75){
              zoomString = "75%";
            }else if(zoomFactor == 1.0){
              zoomString = "100%";
            }else if(zoomFactor == 1.5){
              zoomString = "150%";
            }else if(zoomFactor == 2.0){
              zoomString = "200%";
            }else if(zoomFactor == 5.0){
              zoomString = "500%";
            }else{
              zoomString = "100%";//in case no
match
            }//end else
```

Set pixel color in original image

The code in Listing 5

- creates a new **Color** object based on the values in the text fields of Figure 3 , and
- changes the color of the corresponding pixel in the original image to the new color.

At this point, the color of the pixel in the original image has been modified.

Listing 5 - Set pixel color in original image.

```
          Color newColor = new Color(
redInt,greenInt,blueInt);
pix.getPixel(xIndex,yIndex).setColor(newColor);
```

Create and populate a new PictureExplorer object

Listing 6 begins by disposing of the original **PictureExplorer** object. Then Listing 6 creates a new **PictureExplorer** object containing the modified image.

Finally Listing 6 calls the **setDefaultCloseOperation** on the **JFrame** that houses the **PictureExplorer** object disabling the X-button in the upper-right corner.

Listing 6 - Create and populate a new PictureExplorer object.

Listing 6 - Create and populate a new PictureExplorer object.

```
          //Dispose of the existing explorer
and create a
          // new one.
          explorerFrame.dispose();

          explorer = new PictureExplorer(pix);

          //Get reference to the new frame
          explorerFrame = explorer.getFrame();

explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);
```

Set the state of the PictureExplorer object

You will probably need to study the event handling code in the **PictureExplorer** class to fully understand the remaining code in this discussion.

When you click the image in the **PictureExplorer** object, a **MouseListener** object registered on the **PictureExplorer** object calls the **mousePressed** method belonging to that object

Simulate a physical click

We can simulate a physical click on that image by calling the same method from outside the **PictureExplorer** object.

Call the mousePressed method

Listing 7 calls the **mousePressed** method on the **PictureExplorer** object passing the crosshair coordinates as parameters. Other parameters are also

passed to satisfy the requirements of the **mousePressed** method.

(I will leave it as an exercise for the student to investigate those other parameters.)

Listing 7 - Call the mousePressed method.

```
          //Now set the state of the new
explorer.

          //Simulate a mouse pressed event in
the picture
          // to set the cursor and the text in
the
          // coordinate fields.
          explorer.mousePressed(new
MouseEvent(
                              new JButton("dummy
component"),

MouseEvent.MOUSE_PRESSED,
                              (long)0,
                              0,
                              xIndex,
                              yIndex,
                              0,
                              false));
```

PictureExplorer object handles crosshair cursor

The code in Ericson's **mousePressed** method takes care of all the requirements that result from setting the location of the crosshair cursor, such as setting the coordinate values in the **PictureExplorer** object's text fields shown in Figure 3 .

Set the zoom state

Manually selecting a zoom level from the **Zoom** menu in the **PictureExplorer** object causes the **actionPerformed** method belonging to an **ActionListener** object registered on the **PictureExplorer** object to be executed.

As before, we can simulate the manual selection of a **Zoom** menu value by calling that **actionPerformed** method from outside the object.

Set the zoom state

Listing 8 calls Ericson's **actionPerformed** method, passing the **String** representation of the zoom factor as a parameter.

(I will leave it as an exercise for the student to investigate the other parameters.)

Listing 8 - Set the zoom state.

Listing 8 - Set the zoom state.

```
          //Simulate an action event on the
zoom menu to
          // set the zoom.
          explorer.actionPerformed(new
ActionEvent(
                              explorer,

ActionEvent.ACTION_PERFORMED,
                              zoomString));


        }//end actionPerformed
      }//end newActionListener
    );//end addActionListener
```

Conclusion of discussion

That concludes the discussion of the **ActionListener** object registered on the button labeled **Update Pixel Color** in Figure 1 .

It also concludes the discussion of the program.

Run the program

I encourage you to copy the code from Listing 9 and Listing 10 and download the image file named Prob15.jpg . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: A Pixel Color Editor
- File: Java3130.htm
- Published: 09/11/12
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the source code discussed in this module are shown in Listing 9 and Listing 10 below.

Listing 9 - Complete listing of Prob15.

```
/*File Prob15 Copyright 2012 R.G.Baldwin

The purpose of this program is demonstrate one way
to
change the color of a pixel in a Picture object
that is
encapsulated in a PictureExplorer object.

The pixel to be modified is selected by placing
the
cursor in the PictureExplorer object.
**********************************************
*******/
import java.awt.event.MouseEvent;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;
import javax.swing.JColorChooser;
```

```
public class Prob15{
  public static void main(String[] args){
    new Prob15Runner();
  }//end main method
}//end class Prob15
//===================================================
======//

class Prob15Runner extends JFrame{
  private Prob15Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;
  private JFrame explorerFrame;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new
JPanel();

  private JTextField redField = new
JTextField("000000");
  private JTextField greenField =
                                    new
JTextField("000000");
  private JTextField blueField = new
JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  //Copies of properties of the PictureExplorer
object
  private int xIndex = 0;
  private int yIndex = 0;
  private double zoomFactor = 0;
```

```
  private JButton chooseButton =
                                new JButton("Choose
Color");
  private JButton updateButton =
                           new JButton("Update Pixel
Color");
  //-----------------------------------------------
------//

  public Prob15Runner(){//constructor

    pix = new Picture("Prob15.jpg");
    pix.addMessage("Dick Baldwin",10,20);
    explorer = new PictureExplorer(pix);
    explorerFrame = explorer.getFrame();
    explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);

    //Set the location for the control GUI
    // immediately below the PictureExplorer
object,
    // and set its default close operation.
    setLocation(0,pix.getHeight() + 128);

    setDefaultCloseOperation(

WindowConstants.EXIT_ON_CLOSE);

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
```

```
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    redField.setEditable(false);
    greenField.setEditable(false);
    blueField.setEditable(false);

    colorIndicatorPanel.setBorder(
                              new
LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                    new
Dimension(20,20));

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(chooseButton);
    buttonPanel.add(updateButton);
//    buttonPanel.add(darkerButton);

    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content pane,
adjust to
    // the correct size, and set the tiele.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);

    //---------------------------------------------
------//
```

```
    //Register listeners on the user input
components.
    //-----------------------------------------
------//
    chooseButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
          Color selColor =
JColorChooser.showDialog(
                     chooseButton,"Choose
color",new Color(
                        redInt,greenInt,blueInt));
          if(selColor != null){
            //Don't change the color if the user
cancels
            // out.
            redField.setText("" +
selColor.getRed());
            greenField.setText(
                                "" +
selColor.getGreen());
            blueField.setText("" +
selColor.getBlue());
          }//end if

        }//end action performed
      }//end new ActionListener
    );//end addActionListener
    //-----------------------------------------
------//

    updateButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e)
{
```

```
          //Get properties of PictureExplorer
object.
          xIndex = explorer.getXIndex();
          yIndex = explorer.getYIndex();
          zoomFactor = explorer.getZoomFactor();

          //Save zoom factor as a string.
          String zoomString = "100%";
          if(zoomFactor == 0.25){
            zoomString = "25%";
          }else if(zoomFactor == 0.50){
            zoomString = "50%";
          }else if(zoomFactor == 0.75){
            zoomString = "75%";
          }else if(zoomFactor == 1.0){
            zoomString = "100%";
          }else if(zoomFactor == 1.5){
            zoomString = "150%";
          }else if(zoomFactor == 2.0){
            zoomString = "200%";
          }else if(zoomFactor == 5.0){
            zoomString = "500%";
          }else{
            zoomString = "100%";//in case no match
          }//end else

          Color newColor = new Color(

redInt,greenInt,blueInt);

pix.getPixel(xIndex,yIndex).setColor(newColor);

          //Dispose of the existing explorer and
create a
          // new one.
          explorerFrame.dispose();
```

```
          explorer = new PictureExplorer(pix);
          //Get reference to the new frame
          explorerFrame = explorer.getFrame();
          explorerFrame.setDefaultCloseOperation(

WindowConstants.DO_NOTHING_ON_CLOSE);

          //Now set the state of the new explorer.
          //Simulate a mouse pressed event in the
picture
          // to set the cursor and the text in the
          // coordinate fields.
          explorer.mousePressed(new MouseEvent(
                              new JButton("dummy
component"),

MouseEvent.MOUSE_PRESSED,
                              (long)0,
                              0,
                              xIndex,
                              yIndex,
                              0,
                              false));

          //Simulate an action event on the zoom
menu to
          // set the zoom.
          explorer.actionPerformed(new
ActionEvent(
                                explorer,

ActionEvent.ACTION_PERFORMED,
                                zoomString));


        }//end actionPerformed
      }//end newActionListener
```

```
    );//end addActionListener
    //-----------------------------------------
------//

    //Register a document listener on the red text
field.
    // This listener will respond when the
contents of
    // the text field are modified either by the
program
    // or by the user.
    redField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{
          //Empty method - not needed
        }//end changedUpdate

        public void removeUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <= 255)){
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            redInt = Integer.parseInt(

redField.getText());
            if((redInt >= 0) && (redInt <= 255)){
```

```
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the green
text
    // field. Essentially the same as the above.
    greenField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            greenInt = Integer.parseInt(

greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
```

```
            greenInt = Integer.parseInt(

greenField.getText());
            if((greenInt >= 0) && (greenInt <=
255))
            {
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-----------------------------------------
------//

    //Register a document listener on the blue
text
    // field. Essentially the same as the above.
    blueField.getDocument().addDocumentListener(
      new DocumentListener(){
        public void changedUpdate(DocumentEvent e)
{}

        public void removeUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
```

```
          }//end catch
        }//end removeUpdate

        public void insertUpdate(DocumentEvent e){
          try{
            blueInt = Integer.parseInt(

blueField.getText());
            if((blueInt >= 0) && (blueInt <= 255))
{
              paintColorSwatch();
            }//end if
          }catch(Exception ex){
            //do nothing on exception
          }//end catch
        }//end insertUpdate

      }//end new DocumentListener
    );//end addDocumentListener
    //-------------------------------------------
------//
  }//end constructor
  //-------------------------------------------
------//

  //The purpose of this method is to color a
swatch
  // located next to the RGB color values.
  private void paintColorSwatch(){
    colorIndicatorPanel.setBackground(
                        new
Color(redInt,greenInt,blueInt));
  }//end paintColorSwatch

  //-------------------------------------------
------//
```

```
  //The purpose of this method is to absorb any
exceptions
  // that may be thrown by the parseInt method in
order
  // to avoid having the program abort. In the
event that
  // an exception is thrown, this method simply
returns an
  // int value of 0;
  private int goParseInt(String string){
    int result = 0;
    try{
      result = Integer.parseInt(string);
    }catch(Exception e){
      result = 0;
    }//end catch
    return result;
  }//end goParseInt

  //-----------------------------------------------
------//


}//end class Prob15Runner
```

Listing 10 - Complete listing of modified PictureExplorer class.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;

/*
05/16/12 Modified by Baldwin to add getter methods
to
 cause the following values to be accessible from
```

```
 outside the object:

int xIndex
int yIndex
double zoomFactor
JFrame pictureFrame

Also disabled the call to setDefaultCloseOperation

*/

/**
 * Displays a picture and lets you explore the
picture by displaying the x, y, red,
 * green, and blue values of the pixel at the
cursor when you click a mouse button or
 * press and hold a mouse button while moving the
cursor.  It also lets you zoom in or
 * out.  You can also type in a x and y value to
see the color at that location.
 *
 * Originally created for the Jython Environment
for Students (JES).
 * Modified to work with DrJava by Barbara Ericson
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Keith McDermottt, gte047w@cc.gatech.edu
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class PictureExplorer implements
MouseMotionListener, ActionListener, MouseListener
{

 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;
```

```
//Main gui variables
private JFrame pictureFrame;
private JScrollPane scrollPane;

//information bar variables
private JLabel xLabel;
private JButton xPrevButton;
private JButton yPrevButton;
private JButton xNextButton;
private JButton yNextButton;
private JLabel yLabel;
private JTextField xValue;
private JTextField yValue;
private JLabel rValue;
private JLabel gValue;
private JLabel bValue;
private JLabel colorLabel;
private JPanel colorPanel;

// menu components
private JMenuBar menuBar;
private JMenu zoomMenu;
private JMenuItem twentyFive;
private JMenuItem fifty;
private JMenuItem seventyFive;
private JMenuItem hundred;
private JMenuItem hundredFifty;
private JMenuItem twoHundred;
private JMenuItem fiveHundred;

/** The picture being explored */
private DigitalPicture picture;

/** The image icon used to display the picture */
private ImageIcon scrollImageIcon;

/** The image display */
```

```
 private ImageDisplay imageDisplay;

 /** the zoom factor (amount to zoom) */
 private double zoomFactor;

 /** the number system to use, 0 means starting at
0, 1 means starting at 1 */
 private int numberBase=0;

 /**
  * Public constructor
  * @param picture the picture to explore
  */
 public PictureExplorer(DigitalPicture picture)
 {
   // set the fields
   this.picture=picture;
   zoomFactor=1;

   // create the window and set things up
   createWindow();
 }

//===Methods added by Baldwin on
05/16/12==============//

  //Method to get the xIndex value.
  public int getXIndex(){
    return xIndex;
  }//end getXIndex

  //Method to get the yIndex value.
  public int getYIndex(){
    return yIndex;
  }//end getYIndex

  //Method to get the zoomFactor value.
```

```
  public double getZoomFactor(){
    return zoomFactor;
  }//end getZoomFactor

  //Method to get a reference to the frame
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()

//===End methods added by Baldwin on
05/16/12==========//




 /**
  * Changes the number system to start at one
  */
 public void changeToBaseOne()
 {
   numberBase=1;
 }

 /**
  * Set the title of the frame
  *@param title the title to use in the JFrame
  */
 public void setTitle(String title)
 {
   pictureFrame.setTitle(title);
 }

 /**
  * Method to create and initialize the picture
frame
```

```
  */
 private void createAndInitPictureFrame()
 {
   pictureFrame = new JFrame(); // create the
JFrame
   pictureFrame.setResizable(true);  // allow the
user to resize it
   pictureFrame.getContentPane().setLayout(new
BorderLayout()); // use border layout

   //Disabled by Baldwin on 05/16/12
   //pictureFrame.setDefaultCloseOperation(
   // JFrame.DISPOSE_ON_CLOSE);

   pictureFrame.setTitle(picture.getTitle());
   PictureExplorerFocusTraversalPolicy newPolicy =
new PictureExplorerFocusTraversalPolicy();

pictureFrame.setFocusTraversalPolicy(newPolicy);

 }

 /**
  * Method to create the menu bar, menus, and menu
items
  */
 private void setUpMenuBar()
 {
   //create menu
   menuBar = new JMenuBar();
   zoomMenu = new JMenu("Zoom");
   twentyFive = new JMenuItem("25%");
   fifty = new JMenuItem("50%");
   seventyFive = new JMenuItem("75%");
   hundred = new JMenuItem("100%");
   hundred.setEnabled(false);
   hundredFifty = new JMenuItem("150%");
```

```
    twoHundred = new JMenuItem("200%");
    fiveHundred = new JMenuItem("500%");

    // add the action listeners
    twentyFive.addActionListener(this);
    fifty.addActionListener(this);
    seventyFive.addActionListener(this);
    hundred.addActionListener(this);
    hundredFifty.addActionListener(this);
    twoHundred.addActionListener(this);
    fiveHundred.addActionListener(this);

    // add the menu items to the menus
    zoomMenu.add(twentyFive);
    zoomMenu.add(fifty);
    zoomMenu.add(seventyFive);
    zoomMenu.add(hundred);
    zoomMenu.add(hundredFifty);
    zoomMenu.add(twoHundred);
    zoomMenu.add(fiveHundred);
    menuBar.add(zoomMenu);

    // set the menu bar to this menu
    pictureFrame.setJMenuBar(menuBar);
  }

  /**
   * Create and initialize the scrolling image
   */
  private void createAndInitScrollingImage()
  {
    scrollPane = new JScrollPane();

    BufferedImage bimg =
picture.getBufferedImage();
    imageDisplay = new ImageDisplay(bimg);
    imageDisplay.addMouseMotionListener(this);
```

```
   imageDisplay.addMouseListener(this);
   imageDisplay.setToolTipText("Click a mouse
button on a pixel to see the pixel information");
   scrollPane.setViewportView(imageDisplay);
   pictureFrame.getContentPane().add(scrollPane,
BorderLayout.CENTER);
 }

 /**
  * Creates the JFrame and sets everything up
  */
 private void createWindow()
 {
   // create the picture frame and initialize it
   createAndInitPictureFrame();

   // set up the menu bar
   setUpMenuBar();

   //create the information panel
   createInfoPanel();

   //creates the scrollpane for the picture
   createAndInitScrollingImage();

   // show the picture in the frame at the size it
needs to be
   pictureFrame.pack();
   pictureFrame.setVisible(true);
 }

 /**
  * Method to set up the next and previous buttons
for the
  * pixel location information
  */
 private void setUpNextAndPreviousButtons()
```

```
 {
   // create the image icons for the buttons
   Icon prevIcon = new
ImageIcon(SoundExplorer.class.getResource("leftArr
ow.gif"),
                                  "previous
index");
   Icon nextIcon = new
ImageIcon(SoundExplorer.class.getResource("rightAr
row.gif"),
                                  "next index");
   // create the arrow buttons
   xPrevButton = new JButton(prevIcon);
   xNextButton = new JButton(nextIcon);
   yPrevButton = new JButton(prevIcon);
   yNextButton = new JButton(nextIcon);

   // set the tool tip text
   xNextButton.setToolTipText("Click to go to the
next x value");
   xPrevButton.setToolTipText("Click to go to the
previous x value");
   yNextButton.setToolTipText("Click to go to the
next y value");
   yPrevButton.setToolTipText("Click to go to the
previous y value");

   // set the sizes of the buttons
   int prevWidth = prevIcon.getIconWidth() + 2;
   int nextWidth = nextIcon.getIconWidth() + 2;
   int prevHeight = prevIcon.getIconHeight() + 2;
   int nextHeight = nextIcon.getIconHeight() + 2;
   Dimension prevDimension = new
Dimension(prevWidth,prevHeight);
   Dimension nextDimension = new
Dimension(nextWidth, nextHeight);
   xPrevButton.setPreferredSize(prevDimension);
```

```
   yPrevButton.setPreferredSize(prevDimension);
   xNextButton.setPreferredSize(nextDimension);
   yNextButton.setPreferredSize(nextDimension);

   // handle previous x button press
   xPrevButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt)
{
       xIndex--;
       if (xIndex < 0)
         xIndex = 0;
       displayPixelInformation(xIndex,yIndex);
     }
   });

   // handle previous y button press
   yPrevButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt)
{
       yIndex--;
       if (yIndex < 0)
         yIndex = 0;
       displayPixelInformation(xIndex,yIndex);
     }
   });

   // handle next x button press
   xNextButton.addActionListener(new
ActionListener() {
     public void actionPerformed(ActionEvent evt)
{
       xIndex++;
       if (xIndex >= picture.getWidth())
         xIndex = picture.getWidth() - 1;
       displayPixelInformation(xIndex,yIndex);
```

```
      }
    });

    // handle next y button press
    yNextButton.addActionListener(new
ActionListener() {
      public void actionPerformed(ActionEvent evt)
{
        yIndex++;
        if (yIndex >= picture.getHeight())
          yIndex = picture.getHeight() - 1;
        displayPixelInformation(xIndex,yIndex);
      }
    });
  }

  /**
   * Create the pixel location panel
   * @param labelFont the font for the labels
   * @return the location panel
   */
  public JPanel createLocationPanel(Font labelFont)
{

    // create a location panel
    JPanel locationPanel = new JPanel();
    locationPanel.setLayout(new FlowLayout());
    Box hBox = Box.createHorizontalBox();

    // create the labels
    xLabel = new JLabel("X:");
    yLabel = new JLabel("Y:");

    // create the text fields
    xValue = new JTextField(Integer.toString(xIndex
+ numberBase),6);
    xValue.addActionListener(new ActionListener() {
```

```
     public void actionPerformed(ActionEvent e) {

displayPixelInformation(xValue.getText(),yValue.ge
tText());
     }
   });
   yValue = new JTextField(Integer.toString(yIndex
+ numberBase),6);
   yValue.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {

displayPixelInformation(xValue.getText(),yValue.ge
tText());
     }
   });

   // set up the next and previous buttons
   setUpNextAndPreviousButtons();

   // set up the font for the labels
   xLabel.setFont(labelFont);
   yLabel.setFont(labelFont);
   xValue.setFont(labelFont);
   yValue.setFont(labelFont);

   // add the items to the vertical box and the
box to the panel
   hBox.add(Box.createHorizontalGlue());
   hBox.add(xLabel);
   hBox.add(xPrevButton);
   hBox.add(xValue);
   hBox.add(xNextButton);
   hBox.add(Box.createHorizontalStrut(10));
   hBox.add(yLabel);
   hBox.add(yPrevButton);
   hBox.add(yValue);
   hBox.add(yNextButton);
```

```
    locationPanel.add(hBox);
    hBox.add(Box.createHorizontalGlue());

    return locationPanel;
  }

  /**
   * Create the color information panel
   * @param labelFont the font to use for labels
   * @return the color information panel
   */
  private JPanel createColorInfoPanel(Font
labelFont)
  {
    // create a color info panel
    JPanel colorInfoPanel = new JPanel();
    colorInfoPanel.setLayout(new FlowLayout());

    // get the pixel at the x and y
    Pixel pixel = new Pixel(picture,xIndex,yIndex);

    // create the labels
    rValue = new JLabel("R: " + pixel.getRed());
    gValue = new JLabel("G: " + pixel.getGreen());
    bValue = new JLabel("B: " + pixel.getBlue());

    // create the sample color panel and label
    colorLabel = new JLabel("Color at location: ");
    colorPanel = new JPanel();
    colorPanel.setBorder(new
LineBorder(Color.black,1));

    // set the color sample to the pixel color
    colorPanel.setBackground(pixel.getColor());

    // set the font
    rValue.setFont(labelFont);
```

```
   gValue.setFont(labelFont);
   bValue.setFont(labelFont);
   colorLabel.setFont(labelFont);
   colorPanel.setPreferredSize(new
Dimension(25,25));

   // add items to the color information panel
   colorInfoPanel.add(rValue);
   colorInfoPanel.add(gValue);
   colorInfoPanel.add(bValue);
   colorInfoPanel.add(colorLabel);
   colorInfoPanel.add(colorPanel);

   return colorInfoPanel;
 }

 /**
  * Creates the North JPanel with all the pixel
location
  * and color information
  */
 private void createInfoPanel()
 {
   // create the info panel and set the layout
   JPanel infoPanel = new JPanel();
   infoPanel.setLayout(new BorderLayout());

   // create the font
   Font largerFont = new
Font(infoPanel.getFont().getName(),

infoPanel.getFont().getStyle(),14);

   // create the pixel location panel
   JPanel locationPanel =
createLocationPanel(largerFont);
```

```
   // create the color informaiton panel
   JPanel colorInfoPanel =
createColorInfoPanel(largerFont);

   // add the panels to the info panel

infoPanel.add(BorderLayout.NORTH,locationPanel);

infoPanel.add(BorderLayout.SOUTH,colorInfoPanel);

   // add the info panel

pictureFrame.getContentPane().add(BorderLayout.NOR
TH,infoPanel);
 }

 /**
  * Method to check that the current position is
in the viewing area and if
  * not scroll to center the current position if
possible
  */
 public void checkScroll()
 {
   // get the x and y position in pixels
   int xPos = (int) (xIndex * zoomFactor);
   int yPos = (int) (yIndex * zoomFactor);

   // only do this if the image is larger than
normal
   if (zoomFactor > 1) {

     // get the rectangle that defines the current
view
     JViewport viewport =
scrollPane.getViewport();
     Rectangle rect = viewport.getViewRect();
```

```
      int rectMinX = (int) rect.getX();
      int rectWidth = (int) rect.getWidth();
      int rectMaxX = rectMinX + rectWidth - 1;
      int rectMinY = (int) rect.getY();
      int rectHeight = (int) rect.getHeight();
      int rectMaxY = rectMinY + rectHeight - 1;

      // get the maximum possible x and y index
      int maxIndexX = (int) (picture.getWidth() *
zoomFactor) - rectWidth - 1;
      int maxIndexY = (int) (picture.getHeight() *
zoomFactor) - rectHeight - 1;

      // calculate how to position the current
position in the middle of the viewing
      // area
      int viewX = xPos - (int) (rectWidth / 2);
      int viewY = yPos - (int) (rectHeight / 2);

      // reposition the viewX and viewY if outside
allowed values
      if (viewX < 0)
        viewX = 0;
      else if (viewX > maxIndexX)
        viewX = maxIndexX;
      if (viewY < 0)
        viewY = 0;
      else if (viewY > maxIndexY)
        viewY = maxIndexY;

      // move the viewport upper left point
      viewport.scrollRectToVisible(new
Rectangle(viewX,viewY,rectWidth,rectHeight));
    }
  }

  /**
```

```
  * Zooms in the on picture by scaling the image.
  * It is extremely memory intensive.
  * @param factor the amount to zoom by
  */
 public void zoom(double factor)
 {
   // save the current zoom factor
   zoomFactor = factor;

   // calculate the new width and height and get
an image that size
   int width = (int)
(picture.getWidth()*zoomFactor);
   int height = (int)
(picture.getHeight()*zoomFactor);
   BufferedImage bimg =
picture.getBufferedImage();

   // set the scroll image icon to the new image

imageDisplay.setImage(bimg.getScaledInstance(width
, height, Image.SCALE_DEFAULT));
   imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
   imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
   imageDisplay.revalidate();
   checkScroll();  // check if need to reposition
scroll
 }

 /**
  * Repaints the image on the scrollpane.
  */
 public void repaint()
 {
   pictureFrame.repaint();
```

```
 }

 //****************************************//
 //                Event Listeners         //
 //****************************************//

 /**
  * Called when the mouse is dragged (button held
down and moved)
  * @param e the mouse event
  */
 public void mouseDragged(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method to check if the given x and y are in
the picture
  * @param x the horiztonal value
  * @param y the vertical value
  * @return true if the x and y are in the picture
and false otherwise
  */
 private boolean isLocationInPicture(int x, int y)
 {
   boolean result = false; // the default is false
   if (x >= 0 && x < picture.getWidth() &&
       y >= 0 && y < picture.getHeight())
     result = true;

   return result;
 }

 /**
  * Method to display the pixel information from
the passed x and y but
```

```
  * also converts x and y from strings
  * @param xString the x value as a string from
the user
  * @param yString the y value as a string from
the user
  */
 public void displayPixelInformation(String
xString, String yString)
 {
   int x = -1;
   int y = -1;
   try {
     x = Integer.parseInt(xString);
     x = x - numberBase;
     y = Integer.parseInt(yString);
     y = y - numberBase;
   } catch (Exception ex) {
   }

   if (x >= 0 && y >= 0) {
     displayPixelInformation(x,y);
   }
 }

 /**
  * Method to display pixel information for the
passed x and y
  * @param pictureX the x value in the picture
  * @param pictureY the y value in the picture
  */
 private void displayPixelInformation(int
pictureX, int pictureY)
 {
   // check that this x and y is in range
   if (isLocationInPicture(pictureX, pictureY))
   {
     // save the current x and y index
```

```
      xIndex = pictureX;
      yIndex = pictureY;

      // get the pixel at the x and y
      Pixel pixel = new
Pixel(picture,xIndex,yIndex);

      // set the values based on the pixel
      xValue.setText(Integer.toString(xIndex  +
numberBase));
      yValue.setText(Integer.toString(yIndex +
numberBase));
      rValue.setText("R: " + pixel.getRed());
      gValue.setText("G: " + pixel.getGreen());
      bValue.setText("B: " + pixel.getBlue());
      colorPanel.setBackground(new
Color(pixel.getRed(), pixel.getGreen(),
pixel.getBlue()));

    }
    else
    {
      clearInformation();
    }

    // notify the image display of the current x
and y
    imageDisplay.setCurrentX((int) (xIndex *
zoomFactor));
    imageDisplay.setCurrentY((int) (yIndex *
zoomFactor));
  }

  /**
   * Method to display pixel information based on a
mouse event
   * @param e a mouse event
```

```
  */
 private void displayPixelInformation(MouseEvent
e)
 {

   // get the cursor x and y
   int cursorX = e.getX();
   int cursorY = e.getY();

   // get the x and y in the original (not scaled
image)
   int pictureX = (int) (cursorX / zoomFactor +
numberBase);
   int pictureY = (int) (cursorY / zoomFactor +
numberBase);

   // display the information for this x and y
   displayPixelInformation(pictureX,pictureY);

 }

 /**
  * Method to clear the labels and current color
and reset the
  * current index to -1
  */
 private void clearInformation()
 {
   xValue.setText("N/A");
   yValue.setText("N/A");
   rValue.setText("R: N/A");
   gValue.setText("G: N/A");
   bValue.setText("B: N/A");
   colorPanel.setBackground(Color.black);
   xIndex = -1;
   yIndex = -1;
 }
```

```
 /**
  * Method called when the mouse is moved with no
buttons down
  * @param e the mouse event
  */
 public void mouseMoved(MouseEvent e)
 {}

 /**
  * Method called when the mouse is clicked
  * @param e the mouse event
  */
 public void mouseClicked(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is pushed
down
  * @param e the mouse event
  */
 public void mousePressed(MouseEvent e)
 {
   displayPixelInformation(e);
 }

 /**
  * Method called when the mouse button is
released
  * @param e the mouse event
  */
 public void mouseReleased(MouseEvent e)
 {
 }
```

```
 /**
  * Method called when the component is entered
(mouse moves over it)
  * @param e the mouse event
  */
 public void mouseEntered(MouseEvent e)
 {
 }

 /**
  * Method called when the mouse moves over the
component
  * @param e the mouse event
  */
 public void mouseExited(MouseEvent e)
 {
 }

 /**
  * Method to enable all menu commands
  */
 private void enableZoomItems()
 {
   twentyFive.setEnabled(true);
   fifty.setEnabled(true);
   seventyFive.setEnabled(true);
   hundred.setEnabled(true);
   hundredFifty.setEnabled(true);
   twoHundred.setEnabled(true);
   fiveHundred.setEnabled(true);
 }

 /**
  * Controls the zoom menu bar
  *
  * @param a the ActionEvent
  */
```

```
public void actionPerformed(ActionEvent a)
{

  if(a.getActionCommand().equals("Update"))
  {
    this.repaint();
  }

  if(a.getActionCommand().equals("25%"))
  {
    this.zoom(.25);
    enableZoomItems();
    twentyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("50%"))
  {
    this.zoom(.50);
    enableZoomItems();
    fifty.setEnabled(false);
  }

  if(a.getActionCommand().equals("75%"))
  {
    this.zoom(.75);
    enableZoomItems();
    seventyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("100%"))
  {
    this.zoom(1.0);
    enableZoomItems();
    hundred.setEnabled(false);
  }

  if(a.getActionCommand().equals("150%"))
```

```
    {
      this.zoom(1.5);
      enableZoomItems();
      hundredFifty.setEnabled(false);
    }

    if(a.getActionCommand().equals("200%"))
    {
      this.zoom(2.0);
      enableZoomItems();
      twoHundred.setEnabled(false);
    }

    if(a.getActionCommand().equals("500%"))
    {
      this.zoom(5.0);
      enableZoomItems();
      fiveHundred.setEnabled(false);
    }
  }

  /**
   * Test Main.  It will ask you to pick a file and
then show it
   */
  public static void main( String args[])
  {
    Picture p = new
Picture(FileChooser.pickAFile());
    PictureExplorer test = new PictureExplorer(p);

  }

  /**
   * Class for establishing the focus for the
textfields
   */
```

```
 private class PictureExplorerFocusTraversalPolicy
                 extends FocusTraversalPolicy {

         /**
          * Method to get the next component for
focus
          */
         public Component
getComponentAfter(Container focusCycleRoot,

Component aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
         }

         /**
          * Method to get the previous component
for focus
          */
          public Component
getComponentBefore(Container focusCycleRoot,
                                          Component
aComponent) {
            if (aComponent.equals(xValue))
              return yValue;
            else
              return xValue;
          }

          public Component
getDefaultComponent(Container focusCycleRoot) {
            return xValue;
         }

         public Component
```

```
getLastComponent(Container focusCycleRoot) {
            return yValue;
        }

        public Component
getFirstComponent(Container focusCycleRoot) {
            return xValue;
        }
    }
}
```

-end-

Java OOP: ITSE 2317 Practice Programs 1
A practice test with solutions for ITSE 2317

ITSE2317 - Java Programming (Intermediate) - Practice Test 1

- Java and Media Library Version Requirements
- Input Image Files
- Solution source code files
- Output Images
- New Classes
- Hints
- Testing Your Programs
- Program Specifications
    - Program 1
    - Program 2
    - Program 3
    - Program 4
    - Program 5
- Miscellaneous Information

Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library .

Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX when such class definitions are provided.

Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Program Specifications

Program 1

Listing 1 - Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image file named Prob01.jpg to produce the graphic output image shown in Figure 1 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

Contrary to the general instructions given above, you may not define any new classes to cause your program to behave as required.

You must copy and modify (if necessary) the media classes named World.java, Turtle.java, and SimpleTurtle.java to cause your program to produce the required output. Don't forget to compile these classes after you modify them.

In addition to the output image, your program must produce the following output on the command- line screen, and must substitute your name for mine wherever my name appears both in the image and on the command-line screen:

```
Dick Baldwin
Picture, filename Prob01.jpg height 274 width 365
Dick Baldwin
Dick Baldwin
Dick Baldwin
Dick Baldwin
**************************************************
*******/

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    World mars = new World(200,250);
    Turtle joe = new Turtle(mars);
    joe.forward();
    Turtle bill = new Turtle(mars);
    bill.moveTo(50,125);
    Turtle sue = new Turtle(mars);
    sue.moveTo(150,125);
    Turtle tom = new Turtle(mars);
    tom.moveTo(100,225);
```

```
  }//end main method
}//end class Prob01
//End program specifications.
```

Figure 1 - Required output images for Prob01.



Program 2

Listing 2 - Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses the class definition shown below and Ericson's media library to produce the graphic output image shown in Figure 2 below.

Click here to download a zip file containing source code for a solution.

Contrary to the general instructions given above, you may not define any new classes to cause your program to behave as required.

You must copy and modify (if necessary) the media classes named Turtle.java, and SimpleTurtle.java to cause your program to produce the

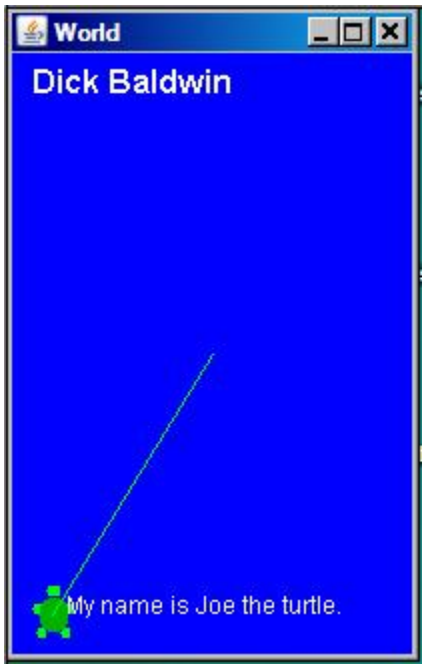
required output. Don't forget to compile these classes after you modify them.

In addition to the output image, your program must produce the following output on the command- line screen, and must substitute your name for mine wherever my name appears both in the image and on the command-line screen:

```
Dick Baldwin
My name is Joe the turtle.
*************************************************
*******/
import java.awt.Color;

public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars,"Joe");
    joe.moveTo(20,280);
    joe.setInfoColor(Color.WHITE);
    joe.setShowInfo(true);
    System.out.println(joe);
  }//end main method
}//end class Prob02
//End program specifications.
```

Figure 2 - Required output images for Prob02.



Program 3

Listing 3 - Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library to produce the graphic output images shown in Figure 3 and Figure 4 below.

Click here to download a zip file containing the source code for a solution.

The image shown in Figure 3 is the image that appears on the screen when the program starts running. The image shown in Figure 4 is what you should see when you click the button at the bottom of the world.

You must copy and modify (if necessary) the media class named World.java, to cause your program to produce the required output with the required behavior. Don't forget to compile that class after you modify it.

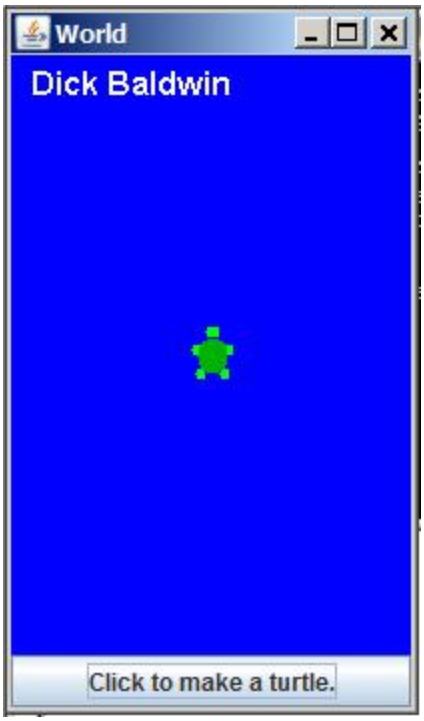
This program adds a JButton object to the SOUTH location of the World object as shown in Figure 3 .

The program initially displays an empty white world. When the user clicks the button, the world's background color changes to blue, a turtle appears in the center of the World, and the student's name appears near the top of the world.

Figure 3 - The first of two required output images for Prob03.



Figure 4 - The second of two required output images for Prob03.



Program 4

Listing 4 - Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses Ericson's media library to produce the graphic output images shown in Figure 5 , Figure 6 , and Figure 7 below.

Click here to download a zip file containing the source code for a solution.

Figure 5 shows the image that appears on the screen when the program starts running. Figure 6 shows what you should see after you have clicked the button at the bottom of the world one time. Figure 6 shows what you should see after you have clicked the button sixteen times.

You must copy and modify the media classes named World.java and SimplePicture.java, to cause your program to produce the required output

with the required behavior. Don't forget to compile those classes after you modify them.

This program adds a JButton object to the SOUTH location of the World object as shown in Figure 5 .

The following description is intended to guide you in writing your program. However, you must run my version of the program and replicate it exactly. I recommend that you run the two programs side-by-side and compare their appearance and behavior each time you click both programs.

The program initially displays an empty white world with a button at the bottom. When the user clicks the button, the world's background color changes to green, a turtle appears in the bottom right of the World, and the student's name appears near the top of the world in blue. The turtle has a blue body and a red shell.

When you click the button again, the background changes to yellow, the student's name changes to red, and the turtle changes to a red body with a blue shell. The turtle turns 90 degrees left and moves forward 100 pixels plus the value of a click counter. As a result, the turtle leaves a blue trail.

On the next click, the colors revert to the same as before, the turtle turns 90 degrees left and moves forward 100 pixels plus the value of the click counter leaving a red trail.

This cycle repeats on each click with the turtle's trail drawing a square spiral of increasing size with red lines on the top and bottom of the spiral and blue lines on the right and left of the spiral.

Figure 5 - Required output image for Prob04.

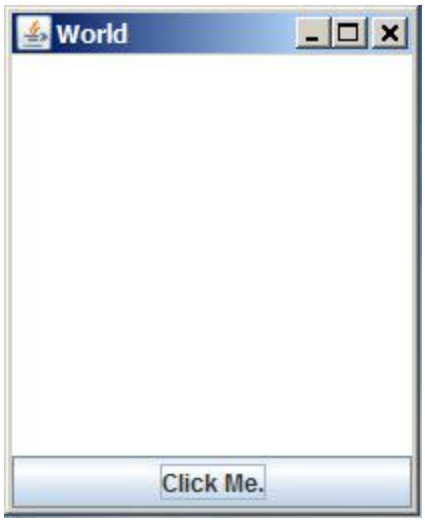


Figure 6 - Required output image for Prob04.

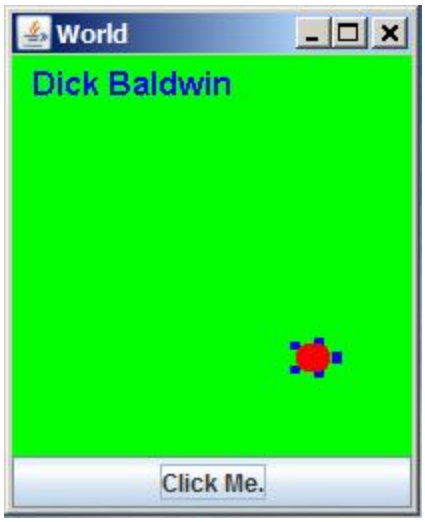
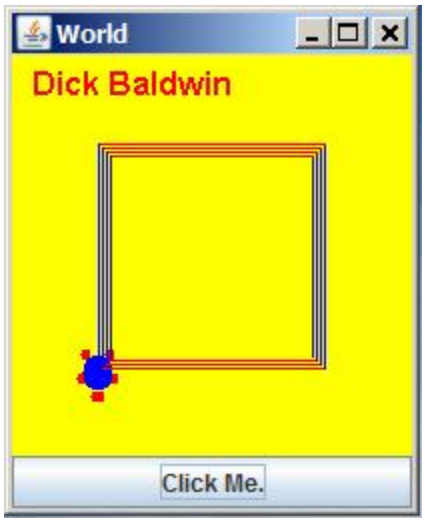


Figure 7 - Required output image for Prob04.



Program 5

Listing 5 - Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library to produce the graphic output image shown in Figure 8 , Figure 9 , and Figure 10 below.

Click here to download a zip file containing the source code for a solution.

Figure 8 shows the image that appears on the screen when the program starts running. Figure 9 shows what you should see after you have entered numeric values into the angle and distance fields and have clicked the Move button. Figure 10 is similar to what you should see after doing the above several times for different numeric values.

You must copy and modify the media class named World.java to cause your program to produce the required output with the required behavior. Don't forget to compile World.java after you modify it.

This program adds two buttons, two labels, and two text fields to form a GUI at the bottom of the World object.

If you enter numeric values into the angle and distance fields and then click the Move button, the turtle will turn by that angle in degrees and move by that distance in pixels.

The program must terminate and return control to the operating system when you click the Quit button.

Note that the GUI at the bottom of the World object is comprised of AWT components instead of Swing components.

Figure 8 - Required output image for Prob05.

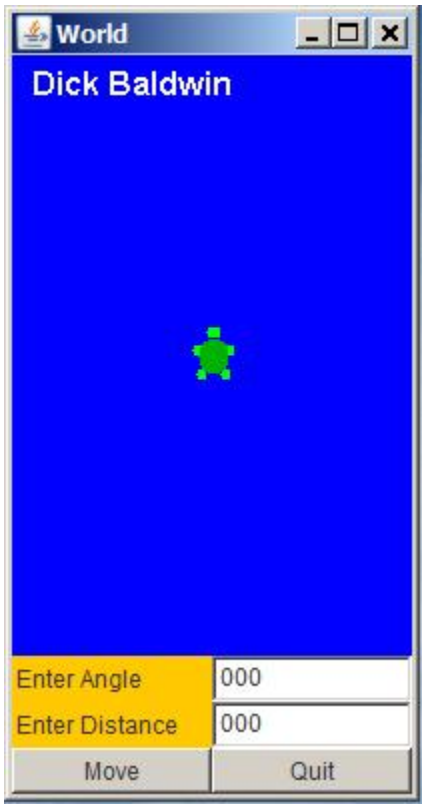


Figure 9 - Required output image for Prob05.

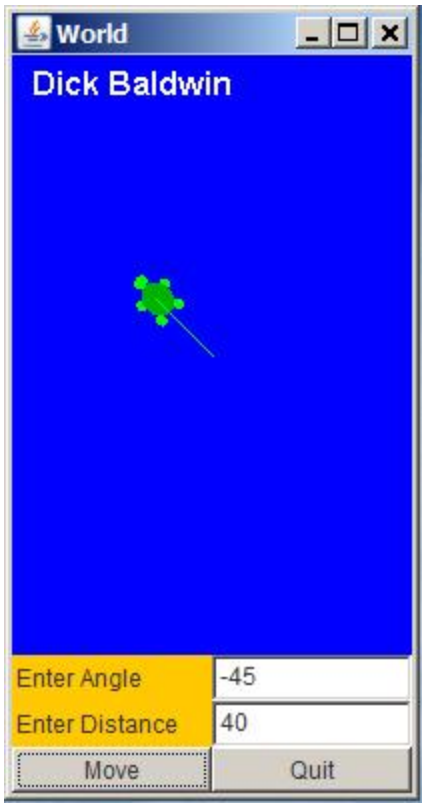
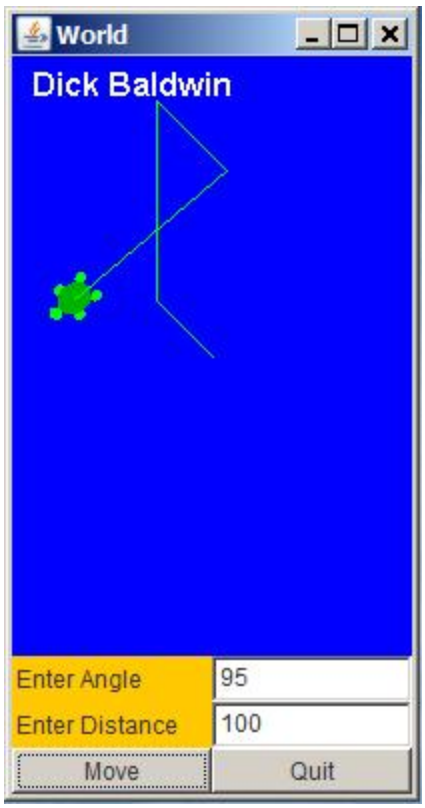


Figure 10 - Required output image for Prob05.



Miscellaneous Information

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: ITSE 2317 Practice Test 1
- File: PracticeTest01.htm
- Published: August 9, 2012
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java OOP: ITSE 2317 Practice Programs 2
A practice test with solutions for ITSE 2317

ITSE2317 - Java Programming (Intermediate) - Practice Test 2

- Java and Media Library Version Requirements
- Input Image Files
- Solution source code files
- Output Images
- New Classes
- Hints
- Testing Your Programs
- Program Specifications
    - Program 1
    - Program 2
    - Program 3
    - Program 4
    - Program 5
- Miscellaneous Information

Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library .

Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX when such class definitions are provided.

Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Program Specifications

Program 1

Listing 1 - Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image files named Prob01a.jpg

and Prob01b.jpg to produce the graphic output images shown in Figure 1 , Figure 2 , and Figure 3 below.

Click here to download a zip file containing the required image files along with the source code for a solution.

Just in case you haven't noticed it, the image in Figure 3 contains a partially transparent image of a butterfly superimposed and centered on the beach image.

In order to write this program, you must modify the class from Ericson's media library named SimplePicture. Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through. The degree of transparency can range from being completely transparent at one extreme to being totally opaque at the other extreme. In this case, the butterfly image is about 37-percent opaque. Don't forget to compile the SimplePicture class after you modify it.

You will probably need to do some outside research in order to write this program. For example, you will need to learn about the following topics and probably some other topics as well:

- Alpha transparency
- BufferedImage objects of TYPE_INT_ARGB
- The representation of a pixel as type int.
- Bit manipulation of pixels.
- The drawImage method of the Graphics class.

In addition to the output images described above, your program must produce the following output on the command-line screen, and must substitute your name for mine wherever my name appears both in the images and on the command-line screen:

```
Dick Baldwin.
Dick Baldwin
Picture, filename Prob01a.jpg height 118 width 100
Picture, filename Prob01b.jpg height 240 width 320
```

```
Picture, filename None height 101 width 77
**********************************************
*******/
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS
DEFINITION.
  public static void main(String[] args){
    Picture[] pictures = new Prob01Runner().run();
    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob01
//End program specifications.
```

Figure 1 - The first of three required output images for Prob01.

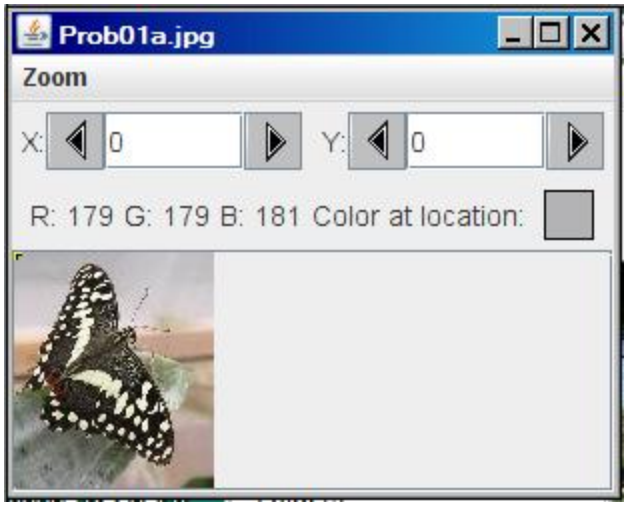


Figure 2 - The second of three required output images for Prob01.

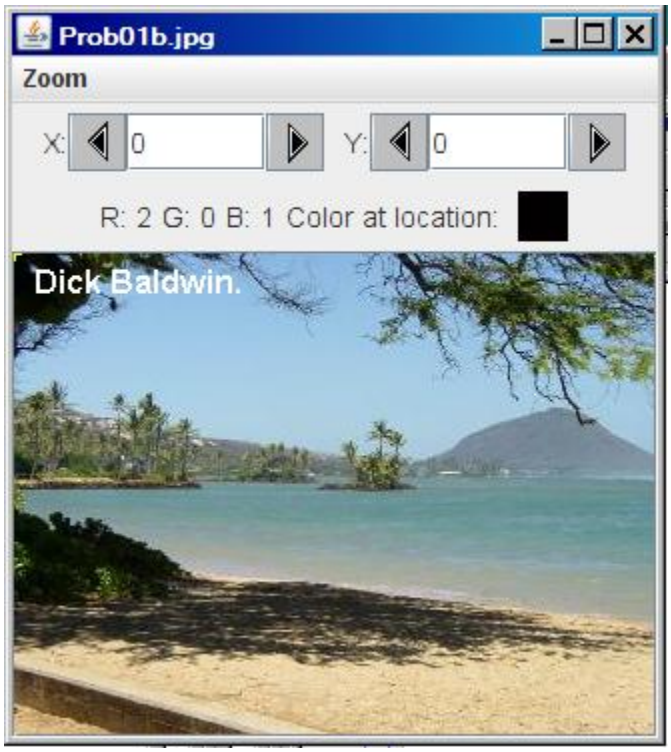
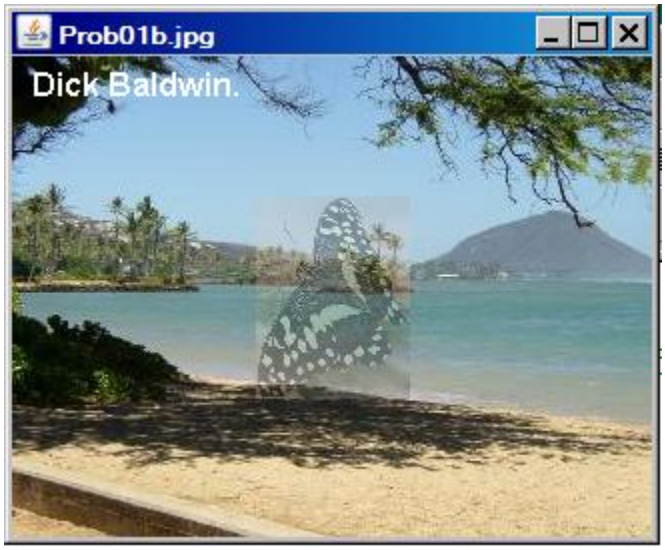


Figure 3 - The third of three required output images for Prob01.



Program 2

Listing 2 - Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses Ericson's media library along with the image files named Prob02a.jpg and Prob02b.jpg to produce the graphic output images shown in Figure 4 below.

Click here to download a zip file containing the required image files along with the source code for a solution.

The top image shown in Figure 4 is a beach scene with a partially opaque butterfly superimposed on the beach scene. The bottom image is a slider that is used to control the percent opacity of the butterfly image.

At startup, the slider is positioned at the 50-percent mark and the opacity of the butterfly is 50 percent.

As you move the slider to the right, the butterfly becomes more opaque, becoming totally opaque when the slider is positioned at 100 percent. As you move the slider to the left, the butterfly becomes less opaque, becoming totally transparent when the slider is positioned at 0 percent.

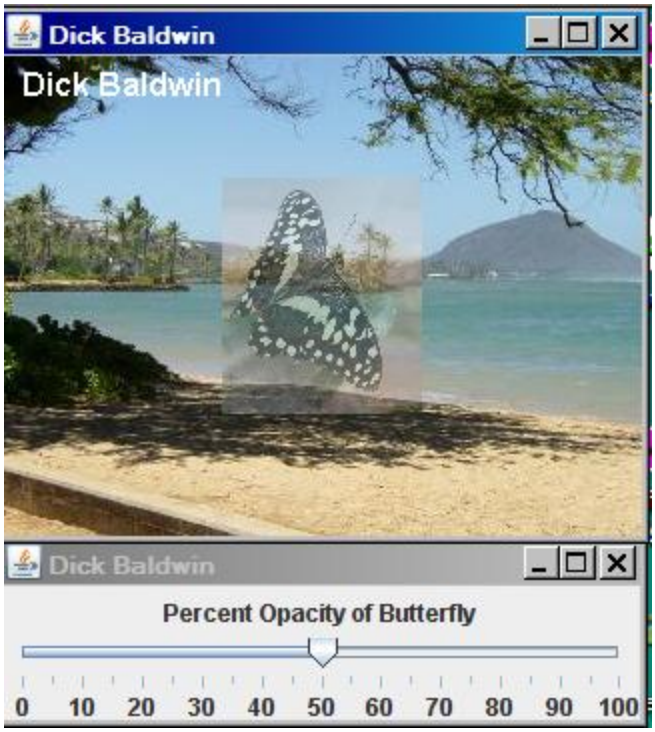
In order to write this program, you must modify the class from Ericson's media library named SimplePicture. Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through.

Your modification must also make it possible to display your name in the dark blue banner at the top of the image of the beach scene.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

In order to improve the responsiveness and memory utilization of the program, you should instantiate all of the Picture objects that the program needs at startup, and should not instantiate additional Picture objects when handling events fired by the slider.

Figure 4 - Required output images for Prob02.



Program 3

Listing 3 - Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library along with the image file named Prob3.jpg to produce the graphic output images shown in Figure 5 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

The top image shown in Figure 5 is an image of a butterfly to which an edge detection algorithm has been applied. The bottom image is a slider that is used to control the edge-detection threshold.

The edge-detection algorithm performs edge detection on a Picture object by rows and also by columns. All edges that are detected by processing adjacent pixels on a row are marked in red. All edges that are detected by processing adjacent pixels on a column are marked in black. If a pixel is determined to be on an edge using both approaches, it ends up being black. If an edge is not detected, the corresponding pixel is marked in white.

At startup, the slider is positioned at the 50-percent mark and the image has been edge-detected using a threshold value of 50. As you move the slider to the right, the threshold increases up to a value of 100, which in turn causes the amount of white area in the image to increase. As you move the slider to the left, the threshold decreases down to a value of zero, which in turn causes the amount of white area in the image to decrease.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

Figure 5 - Required output images for Prob03.



Program 4

Listing 4 - Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses Ericson's media library along with the image files named Prob04a.jpg and Prob04b.jpg to produce the graphic output images shown in Figure 6 and Figure 7 below.

Click here to download a zip file containing the required image files along with the source code for a solution.

The top image shown in Figure 6 is a butterfly image. The image immediately below that one is a slider that is used to control a scale factor that is applied to an image of a beach.

At startup, the slider is positioned at the zero-percent mark *(at the far left)* and the beach image is too small to be seen in the upper-left corner of the butterfly image.

As you move the slider to the right, an image of a beach emerges from the upper-left corner covering the image of the butterfly.

Figure 7 shows the result of moving the slider to the 50-percent mark.

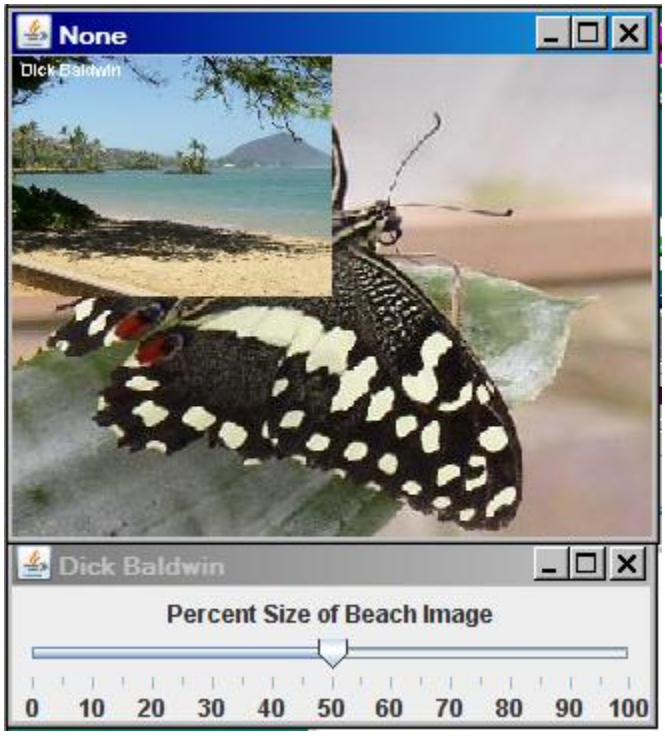
The size of the beach image increases and decreases smoothly as you move the slider back and forth. The upper- left corner of the beach image is always in the upper- left corner of the butterfly image. The butterfly becomes completely covered by the beach image when the slider is positioned at 100 percent *(the far right)* .

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

Figure 6 - Required output images for Prob04.



Figure 7 - Required output image for Prob04.



Program 5

Listing 5 - Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library along with the image file named Prob05.jpg to produce the graphic output images shown in Figure 8 and Figure 9 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

The top image in Figure 8 is a butterfly image. The image of the butterfly can be rotated in its picture by any angle ranging from -360 degrees to +360 degrees.

The image immediately below that one is a slider that is used to control the rotation angle that is applied to the butterfly image.

At startup, the slider is positioned at the zero-degrees mark *(in the center)* and the butterfly image is displayed with no rotation. As you move the slider to the right, the butterfly image rotates clockwise around its center through an angle that can be as large as 360 degrees.

Figure 9 shows the result of moving the slider to the +120-degree mark.

As you move the slider to the left, the butterfly image rotates counter-clockwise around its center through an angle that can be as large as -360 degrees.

The butterfly image rotates smoothly around its center as you move the slider back and forth.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

Figure 8 - Required output image for Prob05.

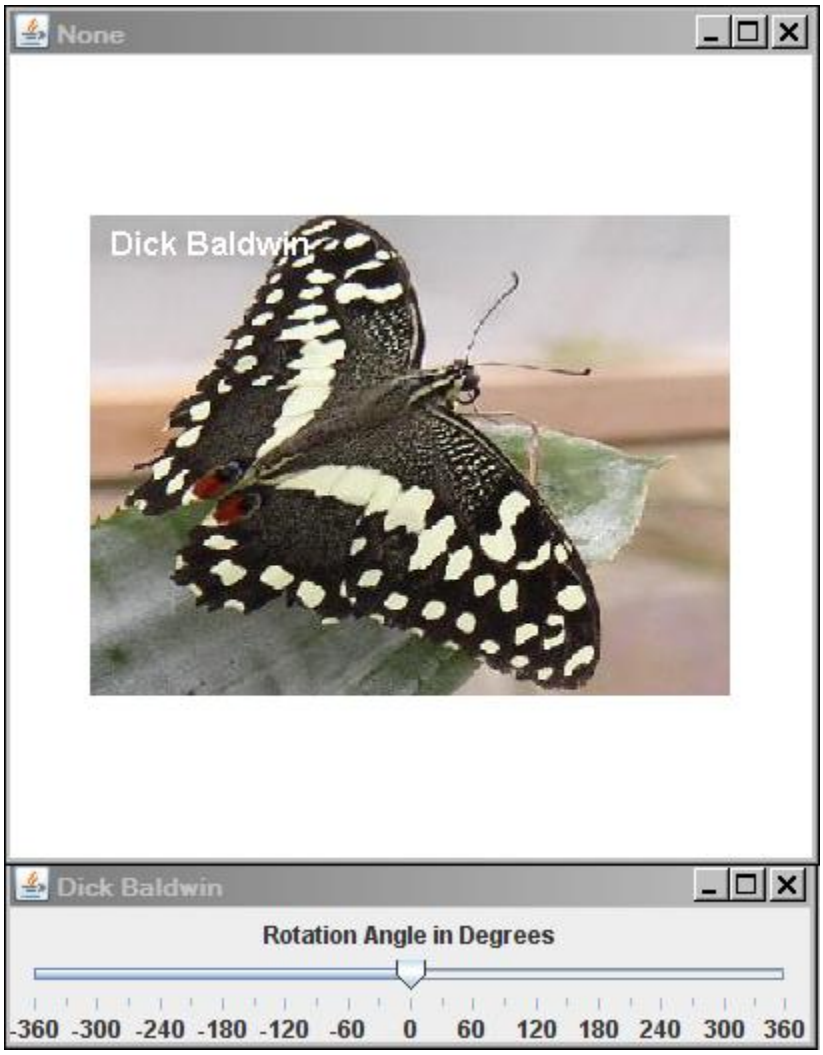
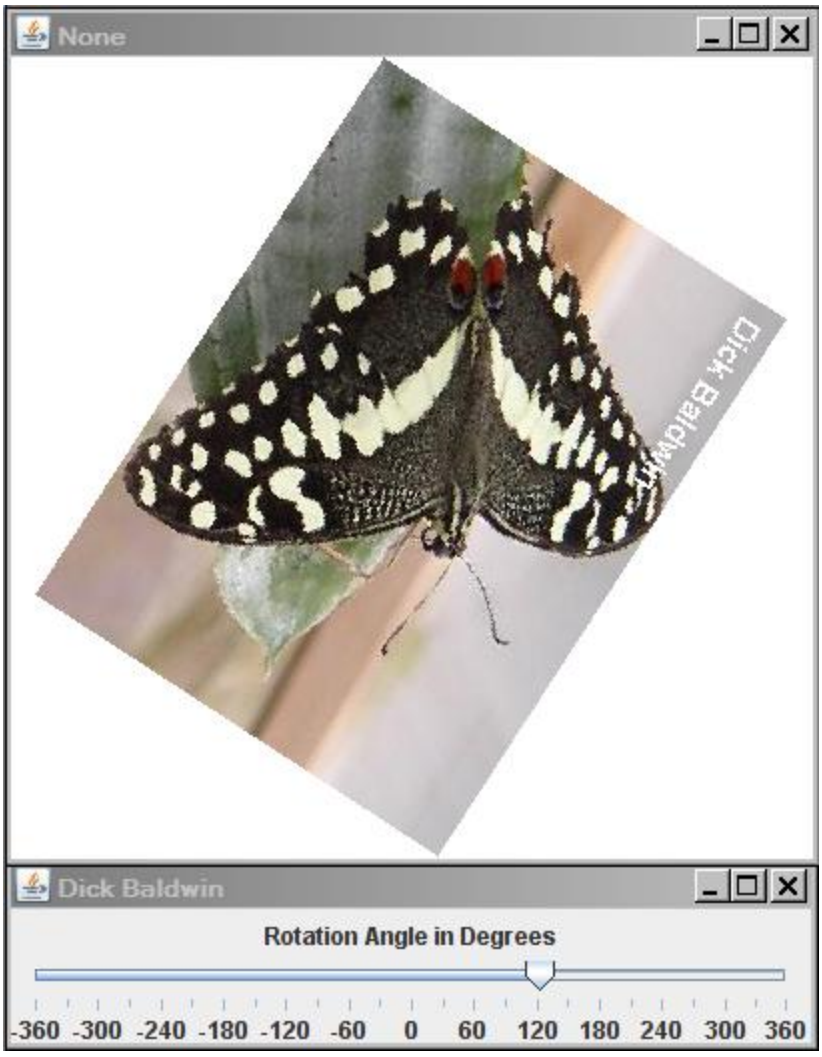


Figure 9 - Required output image for Prob05.



Miscellaneous Information

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: ITSE 2317 Practice Test 2
- File: PracticeTest02.htm
- Published: August 10, 2012
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java OOP: ITSE 2317 Practice Programs 3
A practice test with solutions for ITSE 2317

ITSE2317 - Java Programming (Intermediate) - Practice Test 3

- Java and Media Library Version Requirements
- Input Image Files
- Solution source code files
- Output Images
- Hints
- Testing Your Programs
- Program Specifications
    - Program 1
    - Program 2
    - Program 3
    - Program 4
    - Program 5
- Miscellaneous Information

Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library .

Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.

3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library .

Program Specifications

Program 1

Listing 1 - Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

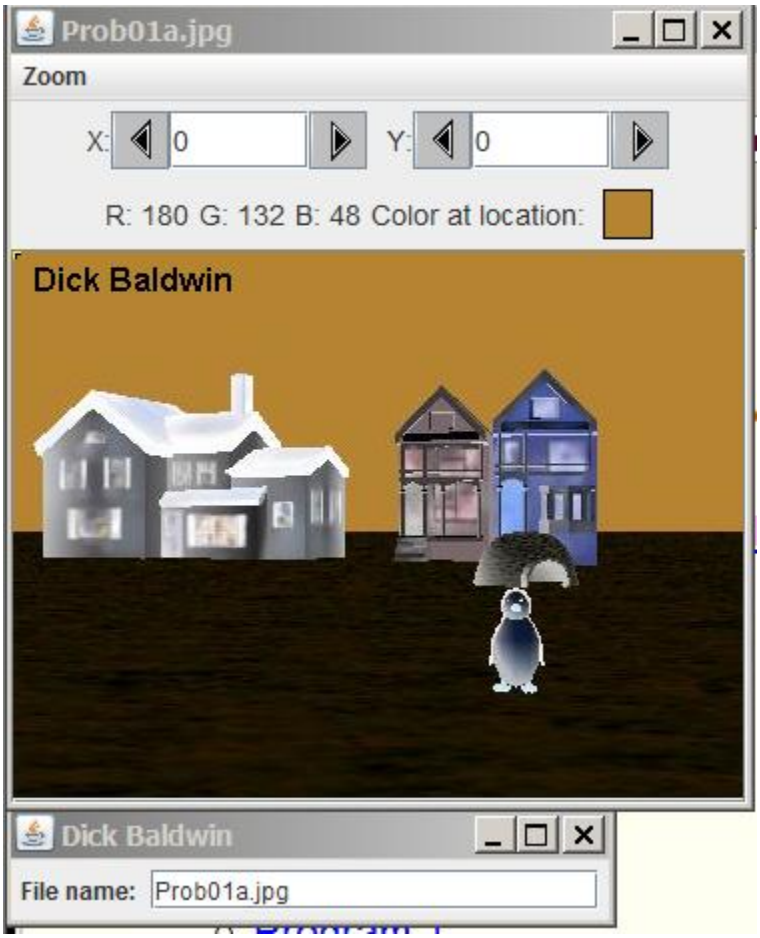
Write a program named Prob01 that uses Ericson's media library along with the image file named Prob01a.jpg to produce the graphic output image shown in Figure 1 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

When the program first starts running, only the GUI containing the text field is visible. When you type the file name into the text field and press Enter, the image of the penguin appears and the GUI moves to a position below the image of the penguin. The program terminates and returns

control to the operating system when you click the X in the upper-right corner of the GUI.

Figure 1 - Required output images for Prob01.



Program 2

Listing 2 - Write the Java application described below.

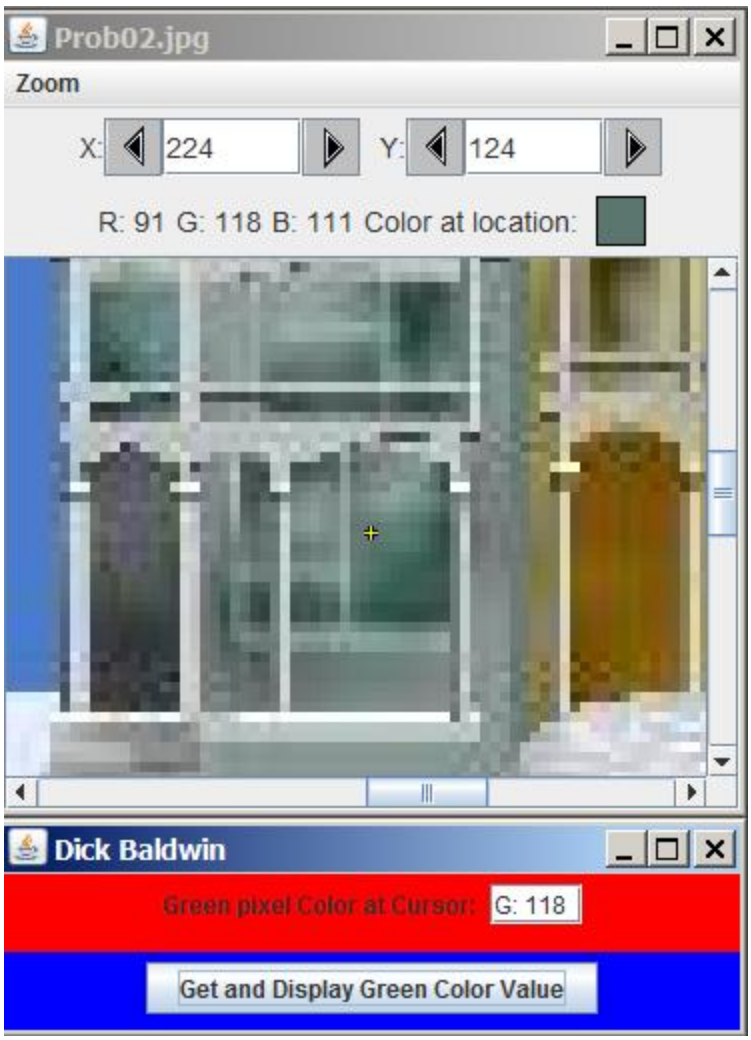
/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses Ericson's media library along with the image file named Prob02.jpg to produce the graphic output images shown in Figure 2 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

Whenever you click the button in the GUI in the lower image, the green color component value at the location of the crosshair cursor in the upper image is displayed in the text field in the GUI.

Figure 2 - Required output images for Prob02.



Program 3

Listing 3 - Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library to produce the graphic output image shown in Figure 3 below.

Click here to download a zip file containing the source code for a solution.

Whenever you click the button in the GUI, the square in the upper right turns from black to yellow with a black border and the RGB color values for the color yellow appear in the three text fields.

Figure 3 - Required output image for Prob03.



Program 4

Listing 4 - Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

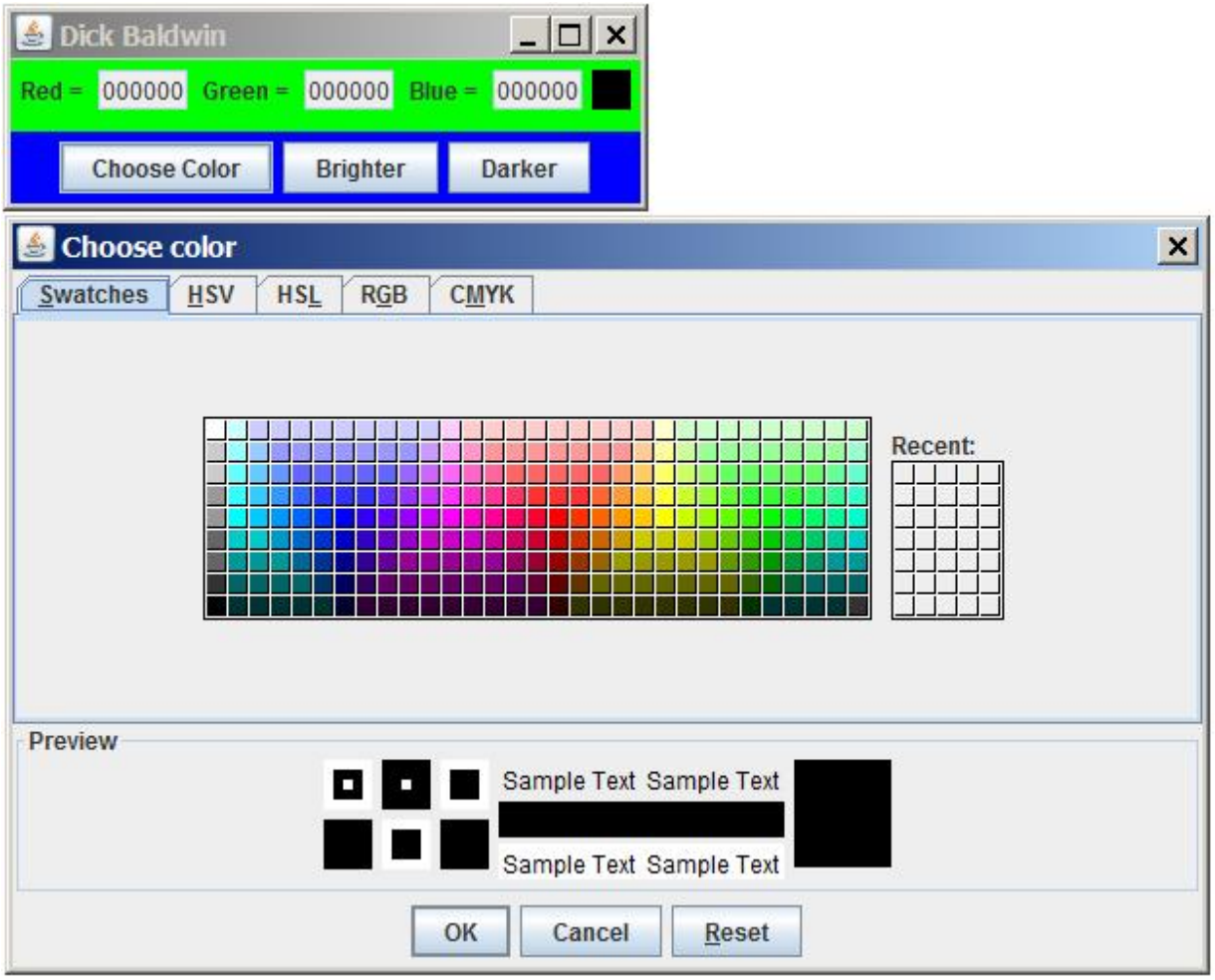
Write a program named Prob04 that uses Ericson's media library to produce the graphic output images shown in Figure 4 below.

Click here to download a zip file containing the source code for a solution.

Whenever you click the "Choose Color" button in the GUI in the upper image, the color chooser dialog shown in the lower image appears. When you select a color and click the OK button, the color chooser dialog disappears and that color appears in the square in the upper right portion of the GUI. In addition, the red, green, and blue color component values for that color appear in the corresponding text fields.

Whenever you click the "Brighter" and "Darker" buttons, the color displayed in the square becomes brighter or darker and the color values in the text fields change accordingly.

Figure 4 - Required output images for Prob04.



Program 5

Listing 5 - Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library along with the image file named Prob05.jpg to produce the graphic output images

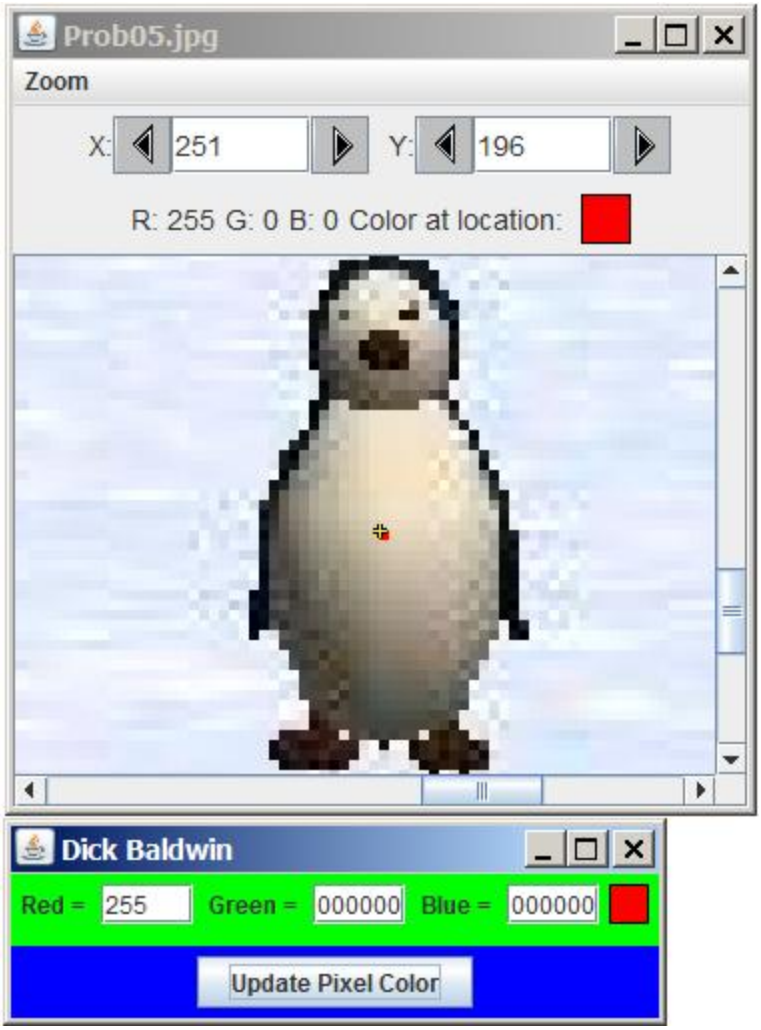
shown in Figure 5 below.

Click here to download a zip file containing the required image file along with the source code for a solution.

The color in the square in the upper right of the GUI in the lower image is governed by the color values in the red, green, and blue text fields.

Whenever you click the button in the GUI, the pixel at the crosshair cursor in the upper image is changed to reflect the color in the square in the GUI. This can best be seen when the upper image is zoomed to 500%.

Figure 5 - Required output image for Prob05.



Miscellaneous Information

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Java OOP: ITSE 2317 Practice Test 3
- File: PracticeTest03.htm
- Published: August 9, 2012
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Jy0037: Review Event Handling
This module contains review questions and answers related to event handling in Java.

Table of Contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 , 17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32 , 33 , 34 , 35 , 36 , 37
- Listings
- Answers
- Miscellaneous

Preface

This module contains a non-exhaustive set of review questions and answers keyed to the material in the Event Handling sub-collection. The questions are heavily weighted toward the tutorials listed in Jy0035: Java OOP: Preface to Event Handling and the seven modules that begin with Java OOP: The AWT and Swing, A Preview in the sub-collection titled **Essence of OOP** .

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

These questions are similar to many of the questions on the online Blackboard test for Event Handling in ITSE 2317. When you take the test in the testing center, you will only have access to the computer screen, a pencil, and a sheet of scratch paper. You might benefit by limiting yourself to only those tools as you study this material to get used to working under those limitations.

Questions

Question 1 .

True or False: AWT is the abbreviation for *Abstract Windows Toolkit* .

Answer 1

Question 2

True or False: GUI is the abbreviation for the Graphical User Interface.

Answer 2

Question 3

True or False: There are two primary packages that are used for **GUI** programming in Java SE 7.

- java.awt.*
- com.sun.java.swing.*

Answer 3

Question 4

True or False: When it was released, Swing was a replacement for the AWT.

Answer 4

Question 5

True or False: A *callback* mechanism is a mechanism where a method in one object asks a method in another object to *"call me back"* or *"notify me"* when an interesting event happens.

Answer 5

Question 6

True or False: Many different objects may ask one object to notify them when the interesting event happens. This is sometimes referred to as *unicasting* .

Answer 6

Question 7

True or False: API is an abbreviation for Application Programming Interface.

Answer 7

Question 8

Given: The event model that has been in use since Java version 1.1 is sometimes called the Delegation Event Model and is sometimes called the JavaBeans Event Model. *(It may be called by other names as well.)* Although there was an earlier model, the Delegation Event Model is the event model to which the questions in this review module apply.

True or False: The event model makes use of **event sources** and **event listeners** .

Answer 8

Question 9

True or False: An event **listener** is an object that has the ability to determine when an interesting event has occurred, and to notify **source** objects of the occurrence of the event.

Answer 9

Question 10

True or False: A **listener** object is an instance of a class (or instance of a subclass of a class) that implements a specific *listener interface* .

Answer 10

Question 11

True or False: A number of listener interfaces are defined where each interface declares the methods appropriate for a specific class of events. Thus, there is natural pairing of classes of events and interface definitions.

Answer 11

Question 12

True or False: There is a class of mouse events that includes most of the events normally associated with mouse action and there is a matching source definition that is used to define a listener class for those events.

Answer 12

Question 13

True or False: A listener object can be registered on a source object to be notified of the occurrence of all events of the specific class for which the listener object is designed.

Answer 13

Question 14

True or False: Once a listener object is registered to be notified of all events of the specific class for which the listener object is designed, the occurrence of such an event will automatically invoke the matching method in the listener object. The code in the body of the method is designed by the programmer to perform the desired action when the event occurs.

Answer 14

Question 15

True or False: In many cases, the same goal can be achieved by either implementing a listener interface or extending a corresponding adapter class.

Answer 15

Question 16

True or False: The program shown below will compile and run successfully. When it starts running, it displays a small window and also displays the following on the command-line screen:

WProc1 windowActivated test msg

WProc1 windowOpened test msg

Whenever the small window loses the focus, the following is displayed on the command-line screen:

WProc1 windowDeactivated test msg

Whenever the small window gains the focus, the following is displayed on the command-line screen:

WProc1 windowActivated test msg

Whenever the user clicks the X-button, the small window disappears and the following is displayed on the command-line screen.

WProc1 windowClosing test msg

WProc1 windowDeactivated test msg

WProc1 windowClosed test msg

Listing 1 . Question 16.

```
/*File Q16.java
**********************************************
********/
import java.awt.*;
import java.awt.event.*;

public class Q16 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
}//end class Q16
//=============================================
=======//
```

```
class GUI{
  public GUI(){//constructor
    //Create a new Frame object
    Frame displayWindow = new Frame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Q16");

    WProc1 winProcCmd1 = new
WProc1(displayWindow);
    displayWindow.addWindowListener(winProcCmd1);
    displayWindow.setVisible(true);
  }//end constructor
}//end class GUI definition
//=============================================
=======//

class WProc1 implements WindowListener{
  Frame displayWindowRef;

  WProc1(Frame windowIn){//constructor
    this.displayWindowRef = windowIn;
  }//end constructor

  public void windowClosed(WindowEvent e){
    System.out.println("WProc1 windowClosed test
msg");
  }//end windowClosed()

  public void windowIconified(WindowEvent e){
    System.out.println("WProc1 windowIconified
test msg");
  }//end windowIconified()

  public void windowOpened(WindowEvent e){
    System.out.println("WProc1 windowOpened test
msg");
```

```
  }//end windowOpened()

  public void windowClosing(WindowEvent e){
    System.out.println("WProc1 windowClosing test
msg");
    displayWindowRef.dispose();//generate
WindowClosed
  }//end windowClosing()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
                       "WProc1 windowDeiconified
test msg");
  }//end windowDeiconified()

  public void windowActivated(WindowEvent e){
    System.out.println("WProc1 windowActivated
test msg");
  }//end windowActivated()

  public void windowDeactivated(WindowEvent e){
    System.out.println(
                      "WProc1 windowDeactivated
test msg");
  }//end windowDeactivated()
}//end class WProc1
//===========================================
=======//
```

Answer 16

Question 17

True or False: The program shown below will compile and run successfully. When it starts running, it displays a small window and doesn't display anything on the command-line screen.

Whenever the small window is iconified to the system try, the following is displayed on the command-line screen:

******** WProc2 windowIconified test msg

Whenever the small window is deiconified from the system try, the following is displayed on the command-line screen:

******** WProc2 windowDeiconified test msg

Nothing happens when the user clicks the X-button. The program does not terminate and does not return control to the operating system.

Listing 2 . Question 17.

```
/*File Q17.java Copyright 1997, R.G.Baldwin
*************************************************
********/

import java.awt.*;
import java.awt.event.*;

public class Q17 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
}//end class Q17
//=============================================
=======//

class GUI{
  public GUI(){//constructor
    //Create a new Frame object
    Frame displayWindow = new Frame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Q17");
```

```
    WProc2 winProcCmd2 = new WProc2();
    displayWindow.addWindowListener(winProcCmd2);
    displayWindow.setVisible(true);

  }//end constructor
}//end class GUI definition
//=============================================
=======//

class WProc2 extends WindowAdapter{

  public void windowIconified(WindowEvent e){
    System.out.println(
              "******** WProc2 windowIconified
test msg");
  }//end windowIconified()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
             "******** WProc2 windowDeiconified
test msg");
  }//end windowDeiconified()

}//end class WProc2
```

Answer 17

Question 18

True or False: The default layout manager for a Frame object is **FlowLayout** .

Answer 18

Question 19

True or False: The program shown below displays a small Frame object on the computer screen. When the user clicks the mouse inside the Frame object, the mouse coordinates are displayed near the mouse pointer.

When the user clicks the X-button in the upper right corner of the Frame, the program terminates and returns control to the operating system.

Listing 3 . Question 19.

```
/*File Q19.java
*/
import java.awt.*;
import java.awt.event.*;

public class Q19 {
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Q19
//-----------------------------------------------
---------

class MyFrame extends Frame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
              "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paint()
}//end class MyFrame
//-----------------------------------------------
---------
```

```
class GUI {
  public GUI(){//constructor
    MyFrame displayWindow = new MyFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Q19");
    displayWindow.setVisible(true);

    displayWindow.addWindowListener(new WProc1());
    displayWindow.addMouseListener(
                                  new
MProc1(displayWindow));
  }//end constructor
}//end class GUI definition
//-----------------------------------------------
---------

class MProc1 extends MouseAdapter{
  MyFrame refToWin; //save a reference to the
window here

  MProc1(MyFrame inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  public void mousePressed(MouseEvent e){
    refToWin.clickX = e.getX();
    refToWin.clickY = e.getY();
    refToWin.repaint();
  }//end mousePressed()
}//end class MProc1
//-----------------------------------------------
---------
class WProc1 extends WindowAdapter{
  public void windowClosed(WindowEvent e){
    System.exit(0);
  }//end windowClosed()
}//end class Wproc1
```

Answer 19

Question 20

True or False: The program shown below will compile and run successfully. When it starts running, it displays a small JFrame window and doesn't display anything on the command-line screen.

Whenever the small window is iconified to the system try, the following is displayed on the command-line screen:

******** WProc2 windowIconified test msg

Whenever the small window is deiconified from the system try, the following is displayed on the command-line screen:

******** WProc2 windowDeiconified test msg

When the user clicks the X-button, the program terminates and returns control to the operating system.

Listing 4 . Question 20.

```
/*File Q20.java
**********************************************
********/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Q20 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a
GUInterface object
  }//end main
```

```
}//end class Q20
//===================================================
=======//

class GUI{
  public GUI(){//constructor
    JFrame displayWindow = new JFrame();
    displayWindow.setSize(300,200);
    displayWindow.setTitle("Q20");
    //Following is available in recent Java
versions
    displayWindow.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
    displayWindow.addWindowListener(new WProc2());
    displayWindow.setVisible(true);

  }//end constructor
}//end class GUI definition
//===================================================
=======//

class WProc2 extends WindowAdapter{
  public void windowIconified(WindowEvent e){
    System.out.println(
              "******** WProc2 windowIconified
test msg");
  }//end windowIconified()

  public void windowDeiconified(WindowEvent e){
    System.out.println(
             "******** WProc2 windowDeiconified
test msg");
  }//end windowDeiconified()
}//end class WProc2
//===================================================
=======//
```

Answer 20

Question 21

True or False: The program shown below displays a small JFrame object on the computer screen. When the user clicks the mouse inside the JFrame object, the mouse coordinates are displayed near the mouse pointer. Old coordinates are erased and new coordinates are displayed with each successive mouse click.

When the user clicks the X-button in the upper right corner of the Frame, the program terminates and returns control to the operating system.

Listing 5 . Question 21.

```
/*File Q21.java
**************************************************
********/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Q21 {
  public static void main(String[] args){
    GUI gui = new GUI();//instantiate a GUI
  }//end main
}//end class Q21
//=================================================
=======//

class MyFrame extends JFrame{
  int clickX;
  int clickY;

  public void paint(Graphics g){
    g.drawString(
```

```
            "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paint()
}//end class MyFrame
//===========================================
=======//

class GUI {
  public GUI(){//constructor
    MyFrame displayWindow = new MyFrame();
    displayWindow.setSize(300,300);
    displayWindow.setTitle("Q21");
    displayWindow.setVisible(true);
    displayWindow.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
    displayWindow.addMouseListener(
                                new
MouseProc(displayWindow));
  }//end constructor
}//end class GUI definition
//===========================================
=======//

class MouseProc extends MouseAdapter{
  MyFrame refToWin; //save a reference to the
source here

  MouseProc(MyFrame inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  public void mousePressed(MouseEvent e){
    refToWin.clickX = e.getX();

    refToWin.clickY = e.getY();
    refToWin.repaint();
```

```
  }//end mousePressed()
}//end class MouseProc
//===========================================
=======//
```

Answer 21

Question 22

True or False: There must always be a one-to-one correspondence between source objects and listener objects.

Answer 22

Question 23

True or False: The program shown below displays two small Frame objects side-by-side on the computer screen. Clicking inside either Frame object causes the mouse coordinates to be displayed in that Frame object relative to the upper-left corner of the Frame object.

Clicking the X-button on either Frame object causes that Frame object to disappear from the computer screen. Clicking the X-button on both Frame objects causes both Frame objects to disappear and causes the program to terminate returning control to the operating system.

Listing 6 . Question 23.

```
/*File Q23.java
*/
//===========================================
=======//

import java.awt.*;
```

```
import java.awt.event.*;

public class Q23 {
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Q23
//===============================================
=======//

class MyFrame extends Frame{
  int xCoor;
  int yCoor;

  MyFrame(){//constructor
    setTitle("Q3");
    setSize(200,200);
  }//end constructor

  public void paint(Graphics g){
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//===============================================
=======//

class GUI {
  public GUI(){//constructor

    MyFrame myFrame1 = new MyFrame();
    myFrame1.setVisible(true);

    MyFrame myFrame2 = new MyFrame();
    myFrame2.setLocation(201,0);
    myFrame2.setVisible(true);
```

```
    WProc1 winProcCmd1 = new WProc1();
    myFrame1.addWindowListener(winProcCmd1);
    myFrame2.addWindowListener(winProcCmd1);

    MouseProc mouseProcCmd = new MouseProc();
    myFrame1.addMouseListener(mouseProcCmd);
    myFrame2.addMouseListener(mouseProcCmd);
  }//end constructor
}//end class GUI definition
//===================================================
=======//

class MouseProc extends MouseAdapter{
  public void mousePressed(MouseEvent e){
    ((MyFrame)e.getComponent()).xCoor = e.getX();
    ((MyFrame)e.getSource()).yCoor = e.getY();
    e.getComponent().repaint();
  }//end mousePressed()
}//end class MouseProc
//===================================================
=======//

class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===================================================
=======//
```

Answer 23

Question 24

True or False: The program shown below displays a small JFrame with a red background. Two overlapping buttons appear in front of the red

background. One button is green and the other button is yellow.

At startup, the green button is in front of and partially hides the yellow button. If you click the frontmost button, the two buttons swap positions front to back. In other words, clicking the green button when it is in front brings the yellow button to the front. Clicking the yellow button when it is in front brings the green button to the front. Clicking the button in the back does not cause the buttons to swap positions.

Clicking the X-button causes the program to terminate and return control to the operating system.

Listing 7 . Question 24.

```
/*File Q24
**********************************************************
********/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Q24 extends JFrame{//subclass JFrame
  JLayeredPane theLayeredPane;
  JButton greenButton;
  JButton yellowButton;

  public static void main(String[] args){
    new Q24();
  }//end main
  //-------------------------------------------------
-------//

  Q24(){//constructor
    theLayeredPane = this.getLayeredPane();
    int frameWidth = 300;
    int frameHeight = 100;
```

```
    JTextField redTextField = new JTextField("");
    redTextField.setBackground(Color.red);

this.getContentPane().add(redTextField,"Center");

    greenButton = new JButton(
                              "greenButton on
Layered Pane");
    greenButton.setBackground(Color.green);
    greenButton.setBounds(10,10,240,40);
    greenButton.addActionListener(new
MyActionListener());
    theLayeredPane.add(greenButton,new
Integer(1));

    yellowButton = new JButton(
                             "yellowButton on
Layered Pane");
    yellowButton.setBackground(Color.yellow);
    yellowButton.setBounds(40,20,240,40);
    yellowButton.addActionListener(new
MyActionListener());
    theLayeredPane.add(yellowButton,new
Integer(-29999));

    //Set title, size, and visibility of JFrame
object.
    this.setTitle("Q24");
    this.setSize(frameWidth,frameHeight);
    this.setVisible(true);


setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }//end constructor

//===========================================
```

```
=====//

  class MyActionListener implements
ActionListener{
    public void actionPerformed(ActionEvent e){
      if(e.getActionCommand().equals(
                            "greenButton on Layered
Pane")){

theLayeredPane.setLayer(greenButton,-29999);
        theLayeredPane.setLayer(yellowButton,1);
      }else{
        theLayeredPane.setLayer(greenButton,1);

theLayeredPane.setLayer(yellowButton,-29999);
      }//end else
    }//end actionPerformed()
  }//end class MyActionListener

//================================================
=====//
}//end class Q24
//================================================
=======//
```

Answer 24

Question 25

True or False: The **MouseMotionListener** interface declares three methods:

- public abstract void mouseDragged(MouseEvent e)
- public abstract void mouseMoved(MouseEvent e)
- public abstract void mousePressed(MouseEvent e)

Answer 25

Question 26

True or False: The **mouseDragged** method is invoked when a mouse button is pressed on a component and then dragged.

Answer 26

Question 27

True or False: The **mouseMoved** method is invoked when the mouse is moved on a component with no buttons down.

Answer 27

Question 28

True or False: This program displays a small Frame object on the computer screen. When you hold down a mouse button and move your mouse pointer into the client area of the Frame, the coordinates of the mouse pointer appear directly above the pointer. As you move the mouse pointer around in the client area while holding the button down, the coordinates of the pointer continue to be displayed above the pointer.

If you release the mouse button while moving the mouse in the client area, the coordinates of the point where you released the button appear and remain there until you press the mouse button, at which time the coordinates of the mouse pointer resume being displayed.

Clicking the X-button terminates the program and returns control to the operating system.

Listing 8 . Question 28.

```
/*File Q28.java  from lesson 92
**********************************************
********/
import java.awt.*;
import java.awt.event.*;

public class Q28 {
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Q28
//=============================================
=========

class MyFrame extends Frame{
  int xCoor;
  int yCoor;

  MyFrame(String name){//constructor
    this.setTitle("Q28");
    this.setSize(300,200);
    this.setName(name);
  }//end constructor

  public void paint(Graphics g){
    g.drawString("" + xCoor + ", " + yCoor, xCoor,
yCoor);
  }//end paint()
}//end class MyFrame
//=============================================
==========

class GUI {
  public GUI(){//constructor
```

```
    MyFrame myFrame1 = new MyFrame("Frame1");
    myFrame1.setVisible(true);

    myFrame1.addWindowListener(new WProc1());

    MyMouseMotionProcessor mouseMotionProc =
                      new
MyMouseMotionProcessor(myFrame1);

myFrame1.addMouseMotionListener(mouseMotionProc);

  }//end constructor
}//end class GUI definition
//=============================================
=========

class MyMouseMotionProcessor extends
MouseMotionAdapter{
  MyFrame refToFrame1; //save references to the
Frame

  MyMouseMotionProcessor(MyFrame inFrame1){
    refToFrame1 = inFrame1;
  }// end constructor

  public void mouseMoved(MouseEvent e){
    refToFrame1.xCoor = e.getX();
    refToFrame1.yCoor = e.getY();
    refToFrame1.repaint();
  }//end mouseMoved()

}//end class MyMouseMotionProcessor
//=============================================
=========

class WProc1 extends WindowAdapter{
  public void windowClosing(WindowEvent e){
```

```
    System.exit(0);
  }//end windowClosing()
}//end class WProc1
//===========================================
=========
```

Answer 28

Question 29

True or False: The program shown below displays a Frame object on the computer screen. The Frame object contains a single large Button object labeled Whistle.

When the user clicks the Whistle button, the program displays the following text on the command-line screen.

I am whistling, Tweet Tweet Tweet

When the user clicks the X-button, the program terminates returning control to the operating system.

Listing 9 . Question 29.

```
/*File Q29.java Copyright 1997, R.G.Baldwin
*****************************************
********/
import java.awt.*;
import java.awt.event.*;

public class Q29 {
  void whistle() {System.out.println(
                      "I am whistling, Tweet Tweet
Tweet");}
  //---------------------------------------
```

```
---------

  static public void main(String[] args){
    Q29 app = new Q29();
    GUI gui = app.new GUI();
  }//end main()
  //-----------------------------------------
---------

  //The GUI class is defined inside the Q29 class
and is
  // an inner-class of Q29.
  class GUI extends Frame{

    public GUI(){//constructor for GUI inner-class
      this.setTitle("Q29");
      Button whistleButton;
      this.add(whistleButton =
                          new
Button("Whistle"),"Center");
      //-------------------------------------
---------
      //Instantiates two anonymous objects of
types
      // ActionListener and WindowAdapter.
registers them
      // for handling
      // events on the Button object and  the
Frame object.
      //Begin statement -------------------------
---------
      whistleButton.addActionListener(
          new ActionListener(){//anonymous class
definition
            public void
actionPerformed(ActionEvent e){
              whistle();//call the whistle()
```

```
method
            }//end actionPerformed()
          }//end ActionListener
        );//end addActionListener()
      //End statement ----------------------------
---------

      //Begin statement --------------------------
---------
      this.addWindowListener(
           new WindowAdapter(){//anonymous class
definition
             public void windowClosing(WindowEvent
e){
               System.exit(0);//terminate the
program
             }//end windowClosing()
           }//end WindowAdapter
         );//end addWindowListener
      //End statement ----------------------------
---------

      //-----------------------------------------
---------
      //Set frame size and make it visible.
      this.setSize(300,100);
      this.setVisible(true);
    }//end GUI constructor
  }//end class GUI
}//end class Q29
//================================================
=========
```

Answer 29

Question 30

True or False: The program shown below displays a small Frame object with a white background. A Button object and a Label object are placed in the Frame. The Label is on the left and the Button is on the right. The two are lined up horizontally and the pair is centered left-to-right in the Frame.

The Button is labeled "Button" and the Label is labeled "Initial Text".

When you click the Button, the text in the Label changes to "Ouch".

When you click the X-button, the program terminates and returns control to the operating system.

Listing 10 . Question 30.

```
/*File Q30.java
**************************************************
********/
import java.awt.*;
import java.awt.event.*;

public class Q30 extends Frame {
  Label myLabel;
  //-----------------------------------------------
-------//
  static public void main(String[] args){
    Q30 app = new Q30();
  }//end main()
  //-----------------------------------------------
-------//

  public Q30(){//constructor
    this.setTitle("Q30");
    this.setLayout(new FlowLayout());
    Button myButton;
    this.add(myButton = new Button("Button"));
    this.add(myLabel = new Label("Initial Text"));
    //---------------------------------------
```

```
-------//

    myButton.addActionListener(
      new //instantiate anonymous object of the
class
        ActionListener(){//anonymous class
definition
          public void actionPerformed(ActionEvent
e){
            myLabel.setText("Ouch");
          }//end actionPerformed()
        }//end ActionListener class definition
      );//end addActionListener() statement

    this.addWindowListener(
         new WindowAdapter(){//anonymous class
definition
           public void windowClosing(WindowEvent
e){
             System.exit(0);//terminate the
program
           }//end windowClosing()
         }//end WindowAdapter
       );//end addWindowListener
    //-----------------------------------------
-------

    this.setSize(300,100);
    this.setVisible(true);
  }//end Q30 constructor
}//end class Q30
//=================================================
=========
```

Answer 30

Question 31

True or False: The program shown below displays a JSlider centered in a JFrame. The JSlider is calibrated to extend from -100 on the left end to 100 on the right end with 0 at the center. There is a tick mark every 10 units and every other tick mark is labeled, including -100, 0, and 100.

When you move the thumb on the slider, the value of the current position of the thumb is displayed on the command-line screen. Old values are not erased when a new value is displayed.

When you click the X-button, the program terminates and returns control to the operating system.

Listing 11 . Question 31.

```
/*File Q31
**************************************************
*******/
import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.awt.BorderLayout;
public class Q31{

  public static void main(String[] args){
    new Q31gui();
  }//end main method
}//end class Q31
//=================================================
======//

class Q31gui extends JFrame{

  private JSlider slider = new JSlider(-100,100);
```

```
  public Q31gui(){//constructor

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(20);
    slider.setMinorTickSpacing(10);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    getContentPane().add(slider);

    setSize(300,100);
    setTitle("Q31");
    setVisible(true);
    //-----------------------------------------
------//

    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          System.out.print(" " +
slider.getValue());
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-----------------------------------------
------//
  }//end constructor
  //-----------------------------------------
------//

}//end class Q31gui
```

Answer 31

Question 32

True or False: The program shown below displays a small JFrame object on the computer screen. Each time the user clicks the mouse inside the JFrame object, the mouse coordinates are displayed near the mouse pointer and the old mouse coordinates from the previous click are erased.

When the user clicks the X-button in the upper right corner of the Frame, the program terminates and returns control to the operating system.

Listing 12 . Question 32.

```
/*File Q32.java Copyright 2002 R.G.Baldwin
*****************************************************
********/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Q32 {
  public static void main(String[] args){
    new Q32gui();
  }//end main
}//end class Q32
//=============================================
=======//

class Q32gui {
  public Q32gui(){//constructor
    JFrame theFrame = new JFrame();
    theFrame.setSize(200,200);
    theFrame.setTitle("Q32");
    DisplaySpace displayWindow = new
DisplaySpace();


theFrame.getContentPane().add(displayWindow,"Cente
```

```
r");
    theFrame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
    theFrame.setVisible(true);

    displayWindow.addMouseListener(
                                  new
MProc1(displayWindow));
  }//end constructor
//=================================================
=======//

class DisplaySpace extends JPanel{

  int clickX;
  int clickY;

  public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.drawString(
               "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paintComponent
}//end class DisplaySpace
//=================================================
=======//

class MProc1 extends MouseAdapter{
  DisplaySpace refToWin;

  MProc1(DisplaySpace inWin){//constructor
    refToWin = inWin;//save ref to window
  }//end constructor

  public void mousePressed(MouseEvent e){
    refToWin.clickX = e.getX();
```

```
    refToWin.clickY = e.getY();
    refToWin.repaint();//display coordinate
information
  }//end mousePressed()
}//end class MProc1
}//end GUI class
//===========================================
=======//
```

Answer 32

Question 33

True or False: The program shown below displays a small JFrame object on the computer screen. Each time the user clicks the mouse inside the JFrame object, the mouse coordinates are displayed near the mouse pointer and the old mouse coordinates from the previous click are erased.

When the user clicks the X-button in the upper right corner of the Frame, the program terminates and returns control to the operating system.

Listing 13 . Question 33.

```
/*File Q33.java
*******************************************
********/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Q33 {
  public static void main(String[] args){
    new Q33gui();
  }//end main
```

```
}//end class Q33
//===========================================
=======//

class Q33gui extends JFrame{
  public Q33gui(){//constructor
    setSize(200,200);
    setTitle("Q33");
    getContentPane().add(new
DisplaySpace(),"Center");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }//end constructor
}//end Q33gui
//===========================================
=======//

class DisplaySpace extends JPanel{

  int clickX;
  int clickY;

  DisplaySpace(){
    addMouseListener(
      new MouseAdapter(){//anonymous class
definition
        public void onMouseEvent(MouseEvent e){
          clickX = e.getX();
          clickY = e.getY();
          repaint();
        }//end onMouseEvent()
      }//end MouseAdapter
    );//end addMouseListener()
  }//end constructor
  //-----------------------------------------
-------//
```

```
  public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.drawString(
              "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paintComponent
}//end class DisplaySpace
//===========================================
=======//
```

Answer 33

Question 34

True or False: The program shown below displays a small Frame object on the computer screen. Each time the user clicks the mouse inside the Frame object, the mouse coordinates are displayed near the mouse pointer and the old mouse coordinates from the previous click are erased.

When the user clicks the X-button in the upper right corner of the Frame, the program terminates and returns control to the operating system.

Listing 14 . Question 34.

```
/*File Q34.java
*/
import java.awt.*;
import java.awt.event.*;

public class Q34 {
  public static void main(String[] args){
    GUI gui = new GUI();
  }//end main
}//end class Q34
```

```
//-----------------------------------------------
-------//

class GUI extends Frame{
  int clickX;
  int clickY;

  public GUI(){//constructor
    setSize(150,150);
    setTitle("Q34");
    setVisible(true);

    addWindowListener(
      new WindowAdapter(){//anonymous class
definition
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end WindowAdapter
    );//end addWindowListener()

    addMouseListener(
      new MouseAdapter(){//anonymous class
definition
        public void mousePressed(MouseEvent e){
          clickX = e.getX();
          clickY = e.getY();
          repaint();
        }//end mousePressed()
      }//end MouseAdapter
    );//end addMouseListener()
  }//end constructor
  //-----------------------------------------------
-------//

  public void paint(Graphics g){
    g.drawString(
```

```
                "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paint

}//end class GUI
//-----------------------------------------------
-------//
```

Answer 34

Question 35

True or False: The program shown below displays a small JFrame object on the computer screen.

When you move your mouse pointer around inside the client area of the frame, without pressing either mouse button, the coordinates of the mouse pointer appear directly above the pointer. In this case, the coordinates are displayed using black characters.

When you move your mouse pointer around inside the client area of the frame, while pressing either mouse button, the coordinates of the mouse pointer appear directly above the pointer. In this case, the coordinates are displayed using red characters.

When you click X-button in the upper right corner of the frame, the program terminates and control is returned to the operating system.

Listing 15 . Question 35.

```
/*File Q35.java
**********************************************
********/

import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;

public class Q35 {
  public static void main(String[] args){
    new Q35gui();
  }//end main
}//end class Q35
//===========================================
=======//

class Q35gui extends JFrame{
  public Q35gui(){//constructor
    setSize(200,200);
    setTitle("Q35");
    getContentPane().add(new
DisplaySpace(),"Center");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }//end constructor
}//end Q35gui
//===========================================
=======//

class DisplaySpace extends JPanel{

  int clickX;
  int clickY;

  DisplaySpace(){
    addMouseMotionListener(
      new MouseMotionAdapter(){//anonymous class
definition
        public void mouseMoved(MouseEvent e){
          setForeground(Color.RED);
          clickX = e.getX();
```

```
          clickY = e.getY();
          repaint();
        }//end mouseMoved
        public void mouseDragged(MouseEvent e){
          setForeground(Color.BLACK);
          clickX = e.getX();
          clickY = e.getY();
          repaint();
        }//end mouseDragged
      }//end MouseAdapter
    );//end addMouseMotionListener()
  }//end constructor
  //-----------------------------------------
-------//

  public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.drawString(
              "" + clickX + ", " + clickY, clickX,
clickY);
  }//end paintComponent
}//end class DisplaySpace
//=========================================
=======//
```

Answer 35

Question 36

True or False: The program shown below displays a small JFrame object containing two JButton objects and a JLabel object. The JLabel has a green background and blue letters. The captions on the two JButton objects are Top and Bottom.

The three objects are displayed vertically in the JFrame. The Top JButton appears at the top of the column of objects. The Bottom JButton appears at

the bottom of the column. The JLabel appears between the two JButton objects.

At startup, the JLabel displays the word Top. When you click the Bottom button, the word in the JLabel changes to Bottom. When you click the Top button, the word in the JLabel changes to Top. When you click X-button on the JFrame, the program terminates and returns control to the operating system.

Listing 16 . Question 36.

```
/*File Q36.java
************************************************
********/
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;

public class Q36{
  public static void main(String[] args){
    JFrame aFrame = new Q36gui();
  }//end main
}//end class Q36
//================================================
=======//

class Q36gui extends JFrame implements
ActionListener{
  JButton button1 = new JButton("Top");
  JButton button2 = new JButton("Bottom");
  JLabel label = new JLabel("Top");
//-----------------------------------------------
```

```
-------//

  public Q36gui(){//constructor
    this.setTitle("Q36");

this.getContentPane().add(button1,BorderLayout.NOR
TH);
    label.setBackground(Color.GREEN);
    label.setForeground(Color.BLUE);
    label.setOpaque(true);

this.getContentPane().add(label,BorderLayout.CENTE
R);

this.getContentPane().add(button2,BorderLayout.SOU
TH);
    button1.addActionListener(this);
    button2.addActionListener(this);
    this.setSize(150,100);
    this.setVisible(true);


this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
);
  }//end constructor
//-----------------------------------------------
-------//

  public void actionPerformed(ActionEvent e){
    if(e.getActionCommand().indexOf("Top") != -1)
      label.setText("Top");
    else
      label.setText("Bottom");
  }//end actionPerformed()

}//end class Q36gui
```

```
//=============================================
=======//
```

Answer 36

Question 37

True or False: The program shown below displays a small Frame object containing two Button objects and a Label object. The Label has a green background and blue letters. The captions on the two Button objects are Top and Bottom.

The three objects are displayed vertically in the Frame. The Top Button appears at the top of the column of objects. The Bottom Button appears at the bottom of the column. The Label appears between the two Button objects.

At startup, the Label displays the word Top. When you click the Bottom button, the word in the Label changes to Bottom. When you click the Top button, the word in the Label changes to Top. When you click X-button on the Frame, the program terminates and returns control to the operating system.

Listing 17 . Question 37.

```
/*File Q37.java Copyright 2002 R.G.Baldwin
*********************************************
********/
import java.awt.Color;
import java.awt.Frame;
import java.awt.Button;
import java.awt.Label;
import java.awt.BorderLayout;

import java.awt.event.ActionListener;
```

```
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;

public class Q37{
  public static void main(String[] args){
    Frame aFrame = new Q37gui();
  }//end main
}//end class Q37
//===========================================
=======//

class Q37gui extends Frame
                implements WindowListener,
ActionListener{
  Button button1 = new Button("Top");
  Button button2 = new Button("Bottom");
  Label label = new Label("Top");
//-------------------------------------------
-------//

  public Q37gui(){//constructor

    this.setTitle("Q37");
    add(button1,BorderLayout.NORTH);
    label.setBackground(Color.GREEN);
    label.setForeground(Color.BLUE);
    add(label,BorderLayout.CENTER);
    add(button2,BorderLayout.SOUTH);
    button1.addActionListener(this);
    button2.addActionListener(this);
    this.setSize(150,100);
    this.setVisible(true);

    this.addWindowListener(this);
  }//end constructor
//-------------------------------------------
```

```
-------//

  public void windowClosing(WindowEvent e){
    //terminate the program when the window is
closed
    System.exit(0);
  }//end windowClosing
//-----------------------------------------------
-------//

  public void windowOpened(WindowEvent e){}//dummy
  public void windowClosed(WindowEvent e){}//dummy
  public void windowIconified(WindowEvent e)
{}//dummy
  public void windowDeiconified(WindowEvent e)
{}//dummy
  public void windowActivated(WindowEvent e)
{}//dummy
  public void windowDeactivated(WindowEvent e)
{}//dummy
//-----------------------------------------------
-------//

  public void actionPerformed(ActionEvent e){
    if(e.getActionCommand().indexOf("Top") != -1)
      label.setText("Top");
    else
      label.setText("Bottom");
  }//end actionPerformed()

}//end class Q37gui
//===============================================
=======//
```

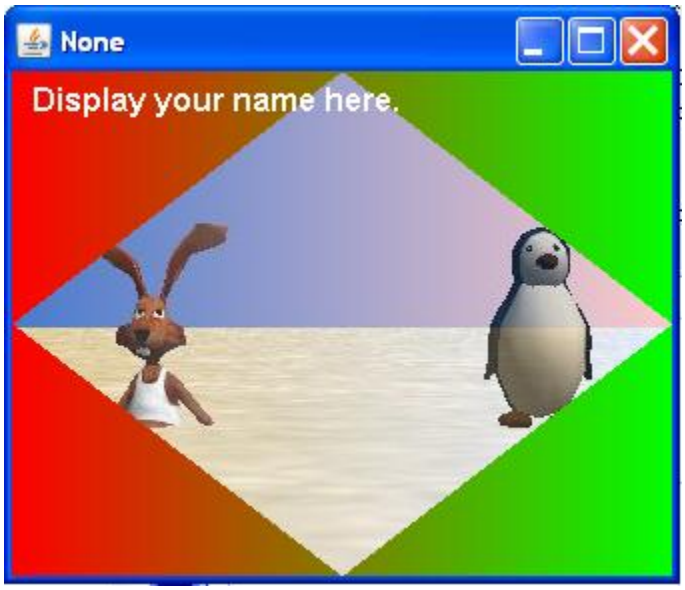
Answer 37

Listings

- Listing 1 . Question 16.
- Listing 2 . Question 17.
- Listing 3 . Question 19.
- Listing 4 . Question 20.
- Listing 5 . Question 21.
- Listing 6 . Question 23.
- Listing 7 . Question 24.
- Listing 8 . Question 28.
- Listing 9 . Question 29.
- Listing 10 . Question 30.
- Listing 11 . Question 31.
- Listing 12 . Question 32.
- Listing 13 . Question 33.
- Listing 14 . Question 34.
- Listing 15 . Question 35.
- Listing 16 . Question 36.
- Listing 17 . Question 37.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the computer screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 37

True. Note that the GUI object is both a source of action events and a listener for action events fired by the buttons.

Back to Question 37

Answer 36

True. Note that the GUI object is both a source of action events and a listener for action events fired by the buttons.

Back to Question 36

Answer 35

False. The red and black colors are reversed. Otherwise, the statement would be true.

Back to Question 35

Answer 34

True. This program uses anonymous inner classes to define and instantiate a mouse event handler and a window event handler.

Back to Question 34

Answer 33

False. The code in the anonymous mouse event handler class calls a method named **onMouseEvent** . It should call a method named **mousePressed** instead, in which case the program would behave as described.

Back to Question 33

Answer 32

True.

Back to Question 32

Answer 31

True.

Back to Question 31

Answer 30

False. The Button and the Label are lined up horizontally with the Button on the Left and the Label on the right. The FlowLayout manager places components from left to right in the order that they are added to the container. The Button was added before the Label. See Java094.

Back to Question 30

Answer 29

True. See Java094.

Back to Question 29

Answer 28

False. The description of the mouse button is reversed. When you move your mouse pointer into the client area of the Frame without pressing a mouse button, the coordinates of the mouse pointer appear directly above the pointer. As you move the mouse pointer around in the client area, the coordinates of the pointer continue to be displayed above the pointer.

If you press one of the mouse buttons while moving the mouse in the client area, the coordinates of the point where you pressed the button appear and remain there until you release the mouse button, at which time the coordinates of the mouse pointer resume being displayed.

Clicking the X-button terminates the program and returns control to the operating system.

See Java092.

Back to Question 28

Answer 27

True. See Java092.

Back to Question 27

Answer 26

True. See Java092.

Back to Question 26

Answer 25

False. The **MouseMotionListener** interface declares only the following two methods. The **mousePressed** method is declared in the **MouseListener** interface. See Java092.

- public abstract void mouseDragged(MouseEvent e)
- public abstract void mouseMoved(MouseEvent e)

Back to Question 25

Answer 24

True. See Java087.

Back to Question 24

Answer 23

False. Clicking the X-button on either Frame object causes both Frame objects to disappear, terminating the program, and returning control to the operating system. See Java082.

Back to Question 23

Answer 22

False. Multiple listener objects can be, and often are registered on a single source object. Similarly, a single listener object can be registered on multiple source objects. In many, but not all such cases, the listener object must be able to identify which source fired the event in order to take the appropriate action. However, in some situations, the appropriate action is independent of the identity of the source. See Java082.

Back to Question 22

Answer 21

False. Although new coordinates are displayed with each mouse click, the old coordinates are not erased. See Java081.

Back to Question 21

Answer 20

True. See Java 081. Note, however that the **setDefaultCloseOperation** method probably did not exist when Java081 was written. When used as shown in the code for this question, that method causes the program to terminate and return control to the operating system when the user clicks

the X-button. Four different default actions can be specified for the X-button:

- DO_NOTHING_ON_CLOSE
- HIDE_ON_CLOSE
- DISPOSE_ON_CLOSE
- EXIT_ON_CLOSE

Back to Question 20

Answer 19

False. In order for this to be true, it would be necessary to override the WindowClosing method instead of the WindowClosed method. See Java080.

Back to Question 19

Answer 18

False: The default layout manager for a Frame object is **BorderLayout** . See Java080.

Back to Question 18

Answer 17

True. See Java080.

Back to Question 17

Answer 16

True. See Java080.

Back to Question 16

Answer 15

True. See Java080.

Back to Question 15

Answer 14

True. See Java080.

Back to Question 14

Answer 13

True. See Java080.

Back to Question 13

Answer 12

False. There is a class of mouse events that includes most of the events normally associated with mouse action and there is a matching **interface** definition that is used to define a listener class for those events. See Java080.

Back to Question 12

Answer 11

True. See Java080.

Back to Question 11

Answer 10

True. See Java080.

Back to Question 10

Answer 9

False. An event **source** is an object that has the ability to determine when an interesting event has occurred, and to notify **listener** objects of the occurrence of the event. See Java080.

Back to Question 9

Answer 8

True. See Java080.

Back to Question 8

Answer 7

True.

Back to Question 7

Answer 6

False. This is *multicasting* . The one-to-one case is often referred to as unicasting. See Java077.

Back to Question 6

Answer 5

True. See Java077.

Back to Question 5

Answer 4

False. It is very important to understand that Swing is an extension of, and not a replacement for the AWT. Therefore, as students, we cannot simply skip over an understanding of the AWT and move on to Swing. The AWT is the foundation for Swing. See Java073.

Back to Question 4

Answer 3

False. com.sun.java.swing.* was the package for Swing in JDK 1.1.6. However, sometime between then and the release of Java SE 7, the package containing Swing components was changed to javax.swing.*

Back to Question 3

Answer 2

True. See Java073.

Back to Question 2

Answer 1

True. See Java073.

Back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Jy0037: Review Event Handling
- File: Jy0037.htm
- Published: 10/30/13
- Revised: 02/06/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com

showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Java4309-Links to JSP Learning Resources
This file contains links to JSP Learning Resources.

Revised: Thu May 12 14:41:22 CDT 2016

This page is included in the following Books:

- *ITSE2317 - Java Programming (Intermediate)*
- *Object-Oriented Programming (OOP) with Java*

Table of contents

- Preface
- Links to Chapters
- Miscellaneous

Preface

- This is the main page for links to JSP content that is part of the course named *ITSE2317 - Java Programming (Intermediate)* at Austin Community College in Austin, TX.
- The material in the chapters is sequential and is designed for progressive study. Most chapters contain material that references concepts and code covered in prior chapters.
- Assignments and the section exam are located in Blackboard.

Links to Chapters

01 - Getting Started with JSP

02 - Web Programming Model

03 - HTML and CSS Fundamentals

04 - JSP Fundamentals

05 - Form Processing with JSP

06 - Working with Java Classes

07 - Cookies with JSP

08 - Sessions with JSP

09 - Assessment Questions

Miscellaneous

Due to the dynamic nature of the material, the content at the links above is periodically updated and the links themselves may change. While you are welcome to print the pages, I recommend bookmarking THIS MAIN PAGE for future reference and NOT the subordinate links. The links to the resources may change but they will always be kept up-to-date on this main page. Since the links may change, by bookmarking this main page, you will have access to the current content and working links.

I hope you enjoy the course.

@author R.L. Martinez, Ph.D.

-end-